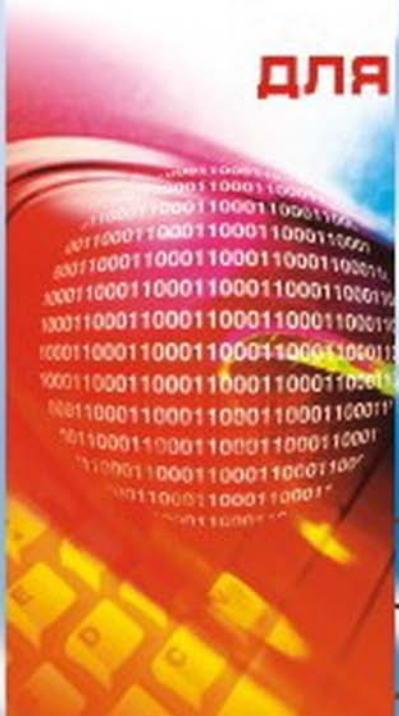
Евгений Марков
Петр Дарахвелидзе

bhv

Delphi 2005 для Win32



- Пользовательский интерфейс и бизнес-логика программирования
- Использование XML
- Современные технологии доступа к данным
- Распределенные многозвенные приложения
- Web-приложения и Web-службы

Наиболее полное руководство

+CD-ROM

В ПОДЛИННИКЕ®

Петр Дарахвелидзе
Евгений Марков

Delphi 2005 для Win32

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.068+800.92Delphi2005
ББК 32.973.26-018.1
Д20

Дарахвелидзе П. Г., Марков Е. П.

Д20 Delphi 2005 для Win32. — СПб.: БХВ-Петербург, 2005. — 1136 с.: ил.

ISBN 5-94157-700-1

Рассмотрены практические аспекты программирования в Borland Delphi 2005 для Win32. Приведено детальное описание объектной модели Delphi, обсуждаются вопросы разработки бизнес-логики приложений. Большое внимание уделено созданию приложения для работы с базами данных. Рассмотрены технологии доступа к данным Borland DataBase Engine, dbExpress, Interbase Express, dbGo, а также создание распределенных приложений баз данных и технология DataSnap. Показана работа с редактором отчетов Rave Report 6.0 и преобразование данных в формат XML. Описана разработка Internet-приложений, распределенные Web-приложения и Web-службы. Все рассматриваемые темы сопровождаются подробными примерами, исходные коды которых находятся на прилагаемом компакт-диске.

Для подготовленных программистов

УДК 681.3.068+800.92Delphi2005
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.08.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 91,59.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.02.953.Д.006421.11.04
от 11.11.2004 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ОАО "Техническая книга"
190005, Санкт-Петербург, Измайловский пр., 29.

ISBN 5-94157-700-1

© Дарахвелидзе П. Г., Марков Е. П., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Часть I. Основы 1

Глава 1. Что и как можно разрабатывать в Delphi 2005 3

Многоязычная среда разработки 3
Язык программирования Delphi 4
Язык программирования C# 6
Язык программирования Visual Basic 6
Программные платформы 7
Компоненты .NET и VCL 8
Какие приложения можно создавать в Delphi 9
Перенос существующих приложений на платформу .NET 11
Резюме 13

Глава 2. Типы данных в Delphi 14

Простые типы 14
Строковый тип 17
Структурные типы 18
Множества 19
Массивы 19
Статические массивы 19
Динамические массивы 21
Многомерные динамические массивы 22
Записи 23
Варианты 24
Резюме 26

Глава 3. Объектно-ориентированное программирование 27

Объект и класс 28
Поля, свойства и методы 31
События 34
Инкапсуляция 38
Наследование 39
Полиморфизм 40
Методы 42
Перегрузка методов 45
Области видимости 47
Объект изнутри 51
Резюме 56

Глава 4. Библиотека визуальных компонентов VCL и ее базовые классы 57

Иерархия базовых классов 57
Класс *TObject* 60
Класс *TPersistent* 62
Класс *TComponent* 63
Базовые классы элементов управления 66
Класс *TControl* 67
Местоположение и размер элемента управления 67
Выравнивание элемента управления 69
Внешний вид элемента управления 70
Связь с родительским элементом управления 72
Класс *TWinControl* 73
Класс *TCustomControl* 75
Класс *TGraphicControl* 76
Резюме 76

Глава 5. Обработка исключительных ситуаций 77

Исключительная ситуация как класс 77
Защитные конструкции языка Delphi 83
Блок *try..except* 83
Блок *try..finally* 86
Использование исключительных ситуаций 88
Протоколирование исключительных ситуаций 91
Коды ошибок в исключительных ситуациях 92
Исключительная ситуация *EAbort* 96
Функция *Assert* 97
Резюме 98

Часть II. Интерфейс и логика приложения 99

Глава 6. Приложение и проект 101

Проект как основа разработки приложения 101
Репозиторий Delphi 108
Класс *TApplication* 109
Атрибуты приложения 116
Обработка сообщений 117
Реакция на действия пользователей 121
Система помощи 121
Резюме 122

Глава 7. Форма 123

Роль формы в приложении 124
Класс *TForm* 127
Создание и уничтожение формы 135
Визуализация формы 137

Атрибуты и стили формы 139
Управление компонентами формы 140
Кадры (Frames). Класс *TFrame* 140
Резюме 141

Глава 8. Элементы управления Win32 и XP 142

Что такое библиотека ComCtl32 142
Многостраничный блокнот — компоненты *TPageControl* и *TTabControl* 144
Компонент *TToolBar* 149
Компонент *TImageList* 153
Компоненты *TTreeView* и *TListView* 155
Календарь 169
Компонент *TMonthCalendar* 169
Компонент *TDateTimePicker* 170
Панель состояния *TStatusBar* 173
Расширенный комбинированный список *TComboBoxEx* 173
Создание нового компонента на базе элементов управления из библиотеки ComCtl32 174
Пользовательский интерфейс Windows XP 186
Манифест Windows XP 187
Компонент *TXPManifest* 189
Включение манифеста Windows XP в ресурсы приложения 190
Визуальные стили и темы оформления 190
Визуальные стили в Delphi 191
Theme API 194
Компоненты настройки цветовой палитры 196
Резюме 197

Глава 9. Меню и действия 198

Редактор меню 199
Как работает меню 200
Главное меню приложения 204
Всплывающее меню 207
Действия. Компонент *TActionList* 207
События, связанные с действиями 209
Свойства, распространяемые на клиентов действия 210
Прочие свойства 212
Стандартные действия 213
Категория *Edit* 215
Категория *Search* 216
Категория *Help* 216
Категория *File* 216
Категория *Dialog* 217
Категория *Window* 217
Категория *Tab* 217

Категория *List* 217
Категория *Internet* 218
Категория *Format* 219
Категория *Dataset* 220
Категория *Tools* 220
Компонент *TActionManager* 220
Изменение и настройка внешнего вида панелей 222
Ручное редактирование коллекций панелей и действий 224
Резюме 227

Глава 10. Списки и коллекции 228

Список строк 229
Класс *TStrings* 229
Класс *TStringList* 230
Список указателей 237
Класс *TList* 238
Пример использования списка указателей 241
Коллекции 245
Класс *TCollection* 246
Класс *TCollectionItem* 247
Резюме 248

Глава 11. Файлы и потоки 249

Использование файловых переменных. Типы файлов 249
Операции ввода-вывода 251
Ввод-вывод с использованием функций Windows API 258
Отложенный (асинхронный) ввод-вывод 262
Контроль ошибок ввода-вывода 264
Атрибуты файла. Поиск файла 265
Потоки 267
Базовые классы *TStream* и *THandleStream* 267
Класс *TFileStream* 269
Класс *TMemoryStream* 271
Класс *TStringStream* 272
Оповещение об изменениях в файловой системе 273
Использование отображаемых файлов 274
Резюме 278

Глава 12. Использование графики 279

Графические инструменты Delphi 279
Класс *TFont* 280
Класс *TPen* 281
Класс *TBrush* 282
Класс *TCanvas* 282
Класс *TGraphic* 288

Класс *TPicture* 290
Класс *TMetafile* 292
Класс *TIcon* 293
Класс *TBitmap* 293
Графический формат JPEG. Класс *TJPEGImage* 298
Компонент *TImage* 301
Использование диалогов для загрузки и сохранения графических файлов 302
Класс *TClipboard* 310
Класс *TScreen* 311
Вывод графики с использованием отображаемых файлов 315
Класс *TAnimate* 318
Резюме 319

Часть III. Использование XML 321

Глава 13. Основы XML 323

Что такое XML 323
Основы синтаксиса XML 326
Пролог 328
Определение 329
Тело документа. Корневой элемент 330
Объектная модель документа 331
Интерфейсы семейства *IDOMNode* 332
Свойства *nodeType*, *nodeName* и *nodeValue* 334
Свойства и методы. управляющие другими вершинами 335
Пространства имен 337
Интерфейс *IDOMDocument* 338
Пример создания приложения, использующего модель DOM 338
Реализация модели DOM в Delphi 340
Модуль *xmldom* 342
Модуль *msxml* 342
Модуль *msxmldom* 342
Модуль *XMLIntf* 342
Модуль *XMLDoc* 342
Интерфейс *IXMLNode* и его отличия от стандарта DOM 343
Взаимосвязь между всеми интерфейсами 345
Загрузка XML 347
Асинхронная загрузка 349
Функции, создающие экземпляр документа 349
Обработка ошибок анализатора 350
Пример использования интерфейсов *IXMLNode* и *IXMLDocument* 352
Анализатор MSXML, или Microsoft XML Core Services 364
Резюме 371

Глава 14. Преобразование данных в формате XML 373

Преобразование данных в формате XML 373
Схема преобразования данных XML 374
Формат пакета данных Delphi 375
Инструментарий преобразования данных XML 376
Утилита XML Mapper 377
Выбор исходного файла 378
Создание пакета данных и документа XML и сохранение преобразованных данных 379
Связывание элементов XML и полей пакета данных 380
Создание трансформационного файла и преобразование данных 381
Резюме 382

Часть IV. Приложения баз данных 383

Глава 15. Архитектура приложений баз данных 385

Как работает приложение баз данных 387
Модуль данных 389
Подключение набора данных 391
Настройка компонента *TDataSource* 393
Отображение данных 395
Резюме 397

Глава 16. Набор данных 398

Абстрактный набор данных 400
Стандартные компоненты 405
Компонент таблицы 406
Компонент запроса 408
Компонент хранимой процедуры 411
Индексы в наборе данных 412
Механизм подключения индексов 413
Список описаний индексов 413
Описание индекса 414
Использование описаний индексов 415
Параметры запросов и хранимых процедур 417
Класс *TParams* 420
Класс *TParam* 421
Состояния набора данных 423
Резюме 426

Глава 17. Поля и типы данных 427

Объекты полей 427
Статические и динамические поля 430
Класс *TField* 432

Виды полей 436
Поля синхронного просмотра 437
Вычисляемые поля 439
Внутренние вычисляемые поля 440
Агрегатные поля 440
Объектные поля 442
Типы данных 442
Ограничения 447
Резюме 451

Глава 18. Управление данными 452

Связанные таблицы 452
Отношение "один ко многим" 453
Отношение "многие ко многим" 455
Поиск данных 456
Поиск по индексам 456
Поиск в диапазоне 458
Поиск по произвольным полям 458
Фильтры 459
Быстрый переход к помеченным записям 461
Диапазоны 464
Резюме 465

Глава 19. Компоненты отображения данных 466

Классификация компонентов отображения данных 466
Табличное представление данных 468
Компонент *TDBGrid* 468
Компонент *TDBCtrlGrid* 479
Навигация по набору данных 481
Представление отдельных полей 484
Компонент *TDBText* 485
Компонент *TDBEdit* 485
Компонент *TDBCheckBox* 486
Компонент *TDBRadioGroup* 486
Компонент *TDBListBox* 487
Компонент *TDBComboBox* 487
Компонент *TDBMemo* 487
Компонент *TDBImage* 488
Компонент *TDBRichEdit* 488
Синхронный просмотр данных 489
Механизм синхронного просмотра 490
Компонент *TDBLookupListBox* 492
Компонент *TDBLookupComboBox* 492
Графическое представление данных 493
Резюме 495

Часть V. Технологии доступа к данным 497

Глава 20. Процессор баз данных Borland Database Engine 499

Архитектура и функции BDE 500
Псевдонимы баз данных и настройка BDE 503
Интерфейс прикладного программирования BDE 513
Соединение с источником данных 524
Компоненты доступа к данным 529
Класс *TBDEDataSet* 529
Класс *TDBDataSet* 534
Компонент *TTable* 536
Компонент *TQuery* 542
Компонент *TStoredProc* 545
Резюме 546

Глава 21. Технология dbExpress 547

Доступ к данным dbExpress 548
Драйверы доступа к данным 550
Соединение с сервером баз данных 550
Управление наборами данных 556
Транзакции 558
Использование компонентов наборов данных 560
Класс *TCustomSQLDataSet* 560
Компонент *TSQLDataSet* 563
Компонент *TSQLTable* 563
Компонент *TSQLQuery* 564
Компонент *TSQLStoredProc* 565
Компонент *TSimpleDataSet* 567
Способы редактирования данных 570
Интерфейсы dbExpress 575
Интерфейс *ISQLDriver* 575
Интерфейс *ISQLConnection* 576
Интерфейс *ISQLCommand* 577
Интерфейс *ISQLCursor* 579
Отладка приложений с технологией dbExpress 579
Распространение приложений с технологией dbExpress 582
Резюме 582

Глава 22. Технология InterBase Express 584

Механизм доступа к данным InterBase Express 585
Компонент *TIBDatabase* 585
Компонент *TIBTransaction* 590
Компоненты доступа к данным 594
Область дескрипторов *XSQLDA* 596
Структура *XSQLVAR* 597

Компонент *TIBTable* 599
Компонент *TIBQuery* 599
Компонент *TIBStoredProc* 601
Компонент *TIBDataSet* 601
Компонент *TIBSQL* 604
Обработка событий 607
Информация о состоянии базы данных 609
Компонент *TIBDatabaseInfo* 609
Компонент *TIBSQLMonitor* 611
Резюме 612

Глава 23. Технология dbGo 613

Основы ADO 613
Перечислители 616
Объекты соединения с источниками данных 617
Сессия 617
Транзакции 618
Наборы рядов 618
Команды 619
Провайдеры ADO 620
Технология dbGo 621
Компоненты ADO 621
Механизм соединения с хранилищем данных ADO 622
Компонент *TADOConnection* 622
Настройка соединения 623
Управление соединением 628
Доступ к связанным наборам данных и командам ADO 632
Объект ошибок ADO 634
Транзакции 634
Наборы данных ADO 636
Класс *TCustomADODataSet* 636
Набор данных 637
Курсор набора данных 638
Локальный буфер 639
Состояние записи 640
Фильтрация 642
Поиск 643
Сортировка 644
Команда ADO 644
Групповые операции 646
Параметры 647
Класс *TParameters* 647
Класс *TParameter* 648
Компонент *TADODataSet* 650
Компонент *TADOTable* 650

Компонент *TADOQuery* 651
Компонент *TADOStoredProc* 652
Команды ADO 652
Объект ошибок ADO 654
Пример приложения ADO 655
Соединение с источником данных 659
Реализация групповых операций 660
Реализация фильтрации 660
Реализация сортировки 660
Резюме 660

Часть VI. Приложения COM и COM+ 663

Глава 24. Механизмы COM 665

Базовые понятия 666
Объект 669
Интерфейс 670
Интерфейс *IUnknown* 672
Сервер 672
Библиотека COM 673
Фабрика класса 675
Библиотека типов 676
Объекты COM в Delphi 676
Класс *TComObject* 677
Класс *TTypedComObject* 679
Интерфейс *IUnknown* в Delphi 679
Тип глобального идентификатора 680
Класс *TInterfacedObject* 680
Фабрика класса в Delphi 681
Класс *TComObjectFactory* 682
Класс *TTypedComObjectFactory* 684
Класс *TComClassManager* 684
Сервер COM в Delphi 685
Класс *TComServer* 685
Библиотека типов в Delphi 687
Простой объект COM в составе внутреннего сервера 689
Создание объекта 690
Создание методов интерфейса 697
Регистрация внутреннего сервера 704
Использование интерфейсов внутреннего сервера COM 704
Резюме 706

Глава 25. Технология Автоматизация 707

Базовые понятия Автоматизации 707
Виды интерфейсов Автоматизации 708

Диспинтерфейсы....708
Дуальные интерфейсы....710
Библиотека типов....710
Маршалинг интерфейсов Автоматизации....710
Объект Автоматизации....711
Сервер Автоматизации....711
Контроллер Автоматизации....711
Реализация Автоматизации в Delphi....711
Интерфейсы Автоматизации....712
Интерфейс *IDispatch*....712
Интерфейсы диспетчеризации....713
Дуальные интерфейсы....714
Объект Автоматизации....714
Класс *TAutoObject*....715
Обработка событий объекта Автоматизации....717
Создание методов-аналогов событий....720
Создание класса-оболочки....720
Инициализация объекта Автоматизации....721
Фабрика класса....722
Класс *TAutoIntfObject*....723
Сервер Автоматизации....724
Контроллер автоматизации....725
Пример приложения Автоматизации....728
Сервер Автоматизации....728
Контроллер Автоматизации....732
Резюме....733

Глава 26. Элементы управления ActiveX....734

Как работают элементы управления ActiveX....735
Контейнеры и регистрация элементов управления ActiveX....738
Предоставление методов....739
События....739
Свойства....739
Страницы свойств....740
Лицензирование....740
Реализация компонентов ActiveX в Delphi....741
Класс компонента ActiveX....742
Фабрика класса компонента ActiveX....743
Среда разработки Delphi как контейнер ActiveX....743
Регистрация компонентов ActiveX....744
Использование готовых компонентов ActiveX....744
Инсталляция готовых элементов управления ActiveX....744
Пример инсталляции элемента управления *TWebBrowser*....746
Резюме....750

Часть VII. Распределенные приложения баз данных.....751

Глава 27. Технология Data Snap.....753

Структура многозвенного приложения в Delphi.....754
Трехзвенное приложение в Delphi.....756
Сервер приложения.....757
Клиентское приложение.....758
Механизм удаленного доступа к данным DataSnap.....759
Компонент *TDCOMConnection*.....759
Компонент *TSocketConnection*.....761
Компонент *TWebConnection*.....764
Вспомогательные компоненты-брокеры соединений.....765
Компонент *TSimpleObjectBroker*.....765
Компонент *TLocalConnection*.....767
Компонент *TSharedConnection*.....768
Компонент *TConnectionBroker*.....769
Резюме.....769

Глава 28. Сервер приложения.....770

Структура сервера приложения.....771
Интерфейс *IAppServer*.....772
Удаленные модули данных.....775
Удаленный модуль данных для сервера Автоматизации.....775
Дочерние удаленные модули данных.....780
Провайдеры данных.....781
Интерфейс *IProviderSupport*.....786
Регистрация сервера приложения.....786
Пример простого сервера приложения.....787
Главная форма сервера приложения.....787
Главный удаленный модуль данных.....788
Дочерний удаленный модуль данных.....789
Регистрация сервера приложения.....791
Резюме.....791

Глава 29. Клиент многозвенного распределенного приложения.....792

Структура клиентского приложения.....793
Компонент *TClientDataSet*.....794
Получение данных от компонента-провайдера.....795
Кэширование и редактирование данных.....797
Управление запросом на сервере.....799
Использование индексов.....800
Сохранение набора данных в файлах.....802
Работа с данными типа BLOB.....803
Представление данных в формате XML.....804

Агрегаты 804
Объекты-агрегаты 805
Агрегатные поля 807
Группировка и использование индексов 809
Вложенные наборы данных 809
Дополнительные свойства полей клиентского набора данных 810
Обработка ошибок 811
Пример "тонкого" клиента 814
Соединение клиента с сервером приложения 816
Наборы данных клиентского приложения 818
Резюме 819

Глава 30. Клиент распределенного приложения на основе данных XML 820

Преобразование данных XML в распределенных приложениях 820
Использование данных XML в распределенных приложениях 824
Данные XML в клиентском наборе данных 826
Данные XML в документе XML 826
Данные XML на странице HTML 827
Пример приложения, использующего данные XML 830
Резюме 833

Часть VIII. Генератор отчетов Rave Reports 6.0 835

Глава 31. Отчеты Rave Reports в приложении 837

Генератор отчетов Rave Reports 6 837
Компоненты Rave Reports и их назначение 838
Отчет в приложении Delphi 839
Компонент отчета *TRvProject* 841
Компонент управления отчетом *TRvSystem* 843
Резюме 849

Глава 32. Визуальная среда создания отчетов 850

Инструментарий визуальной среды создания отчетов 851
Проект отчета 854
Библиотека отчетов 855
Каталог глобальных страниц 856
Словарь просмотра данных 856
Стандартные элементы оформления и их свойства 857
Элементы для представления текста и изображений 858
Графические элементы 859
Штрихкоды 859
Обработка событий 860
Внешние источники данных в отчете 861
Соединение с источником данных и просмотры 861
Безопасность доступа к данным 863

Отображение данных в отчетах 863
Структурные элементы отчета 863
Элементы отображения данных 866
Резюме 867

Глава 33. Разработка, просмотр и печать отчетов 868

Этапы создания отчета и включение его в приложение 869
Простой отчет в визуальной среде Rave Reports 869
Нумерация страниц отчета 870
Использование элемента *FontMaster* 871
Добавление страниц к отчету 871
Отчет в приложении 873
Просмотр и печать отчета 875
Сохранение отчета во внешнем файле 876
Компонент *TRvNDRWriter* 877
Преобразование форматов данных 879
Резюме 880

Глава 34. Отчеты для приложений баз данных 881

Соединения с источниками данных в Rave Reports 882
Соединения с источниками данных в визуальной среде Rave Reports 884
Соединение через драйвер Rave Reports 884
Соединение через компонент приложения Delphi 886
Соединения с источниками данных в приложении 887
Компонент *TRvDataSetConnection* 887
Компоненты, использующие BDE 890
Компонент *TRvCustomConnection* 890
Аутентификация пользователя в отчете 893
Типы отчетов 894
Простой табличный отчет 894
Отчет "один ко многим" 896
Группирующий отчет 898
Использование вычисляемых значений 899
Вычисляемые значения по одному источнику 900
Вычисляемые значения по нескольким источникам 901
Управляющие вычислительные элементы 903
Резюме 904

Часть IX. Основы разработки Internet-приложений 905

Глава 35. Введение в архитектуру сетей 907

Модель OSI 908
Физический уровень 910
Протоколы канального уровня 911

Управление доступом к среде 911
Управление логическим каналом (Logical Link Control) 913
Функции сетевого уровня 914
Транспортный уровень 917
Концепции сессионного уровня, уровней представления и приложений....918
Сессионный (session) уровень 919
Уровень представления....919
Уровень приложения 919
Резюме....920

Глава 36. Криптографическая защита информации в Internet 921

Основные термины и понятия криптографии 922
Секретность....922
Аутентификация 925
Целостность 926
Цифровые подписи, сертификаты и их применение....927
Использование *CryptoAPI* 933
Структура *CryptoAPI* 934
Пример использования *CryptoAPI* – менеджер сертификатов 936
Служба сертификатов Microsoft....941
Протоколы Internet для защищенных соединений 943
Настройка SSL на стороне сервера IIS 6....945
Настройка протокола SSL на клиентской стороне 949
Резюме....951

Часть X. Распределенные Web-приложения и Web-службы....953

Глава 37. Серверные Web-приложения....955

Архитектура серверных Web-приложений 956
Типы серверных Web-приложений 958
Введение в интерфейсы CGI и ISAPI....959
Как работает серверное Web-приложение 961
Классы приложений....962
Действия....963
Запросы и ответы 965
Модули данных....966
Использование отладочного сервера Web App Debugger....967
Перенос тестового приложения на рабочую платформу 968
Особенности отладки приложений ISAPI 968
Резюме....973

Глава 38. Технология Web Broker 974

Обзор технологии Web Broker....975
Структура серверного Web-приложения на основе технологии Web Broker....976

Создание серверного Web-приложения....978
Диспетчеризация запросов. Глобальный объект приложения....979
Web-модули и действия....979
Компоненты-продюсеры....982
Страницы HTML в серверных Web-приложениях....984
Отображение данных....985
Ввод и редактирование данных....987
Cookies....991
Использование баз данных....993
Публикация записей таблиц баз данных....993
Генерация отчетов....995
Редактирование данных....1000
Пример простого серверного Web-приложения....1002
Главное окно приложения....1002
Просмотр таблицы Country....1003
Тестирование приложения....1006
Резюме....1007

Глава 39. Протокол SOAP и Web-службы. Клиентская часть....1008

Почему SOAP?....1008
Delphi и *TSomeConnection*....1009
SOAP — решение проблем....1010
Web-службы....1012
Описание протокола WSDL....1013
Архитектура Web-служб в Delphi....1019
Клиент Web-службы....1019
Генерация интерфейса Web-службы....1020
O, RIO, RIO....1024
Как связаться с нужной службой....1026
Решение коммуникационных проблем....1028
Что такое *InvokeRegistry*. Регистрация интерфейсов и типов данных....1031
Резюме....1034

Глава 40. Серверная часть Web-службы....1035

Назначение и настройки компонентов серверной части....1038
Компонент *TWSDLHTMLPublish*....1039
Компоненты *THTTPSoapDispatcher* и *THTTPSoapPascalInvoker*....1042
Клиент службы *SimpleEcho*....1043
Обработка исключительных ситуаций....1043
Средства разработки для SOAP: подход Microsoft....1046
Краткий обзор архитектуры Web-служб в SOAP Toolkit....1047
Сценарий клиентской части....1049
Создание COM-объекта *SimpleEchoCOM* и публикация Web-службы на его основе....1051
Использование утилиты SOAP Trace....1054
Резюме....1056

Глава 41. Технология WebSnap 1057
Структура приложения WebSnap 1058
Обработка запросов 1060
Web-модули 1060
Модуль данных 1061
Модуль страницы 1062
Модуль приложения 1063
Компоненты уровня приложения 1064
Управление компонентами приложения 1064
Адаптер и глобальная переменная приложения 1066
Информация о пользователях 1067
Сессии 1070
Управление файлами приложения 1071
Компоненты-диспетчеры 1071
Компоненты-продюсеры 1073
Компоненты-адаптеры 1074
Создание приложения WebSnap 1077
Конструирование интерфейса и управление данными 1081
Использование скриптов 1082
Навигация по страницам приложения 1083
Использование полей и действий компонентов-адаптеров 1084
Взаимодействие с базами данных 1087
Специализированные элементы управления 1087
Просмотр данных 1089
Редактирование данных 1091
Обработка ошибок 1091
Аутенификация пользователей 1093
Списки пользователей и сессии 1093
Создание страницы аутенификации 1094
Настройка приложения 1096
Использование XML и XSL 1096
Использование файла XML и шаблона XSL 1097
Использование набора данных и шаблона XSL 1099
Дополнительный компонент *TAdapterXMLBuilder* 1101
Резюме 1102

Приложение. Описание компакт-диска 1103

Предметный указатель 1105

ЧАСТЬ I

Основы

Глава 1

Что и как можно разрабатывать в Delphi 2005

Если задуматься, какой девиз лучше всего отражает сущность среды разработки Delphi 2005, то, по нашему мнению, это будет девиз "универсальность". Конечно же, главным новшеством последних двух версий Delphi является возможность разработки приложений для операционной среды .NET корпорации Microsoft. Однако если не полениться и выполнить команду меню **Help | About Borland Developer Studio**, то мы увидим, что официальное название программы — Borland Developer Studio for Microsoft Windows. Это означает, что здесь можно создавать приложения для любых операционных систем Windows, широко используемых в настоящее время.

И действительно, в Delphi 2005 можно программировать для Win32 и .NET, вести разработку на двух языках программирования и при этом создавать приложения многих типов (базы данных, ASP.NET, консольные приложения и т. д.). Кроме этого, здесь можно работать и с приложениями Visual Basic для .NET.

Две операционные среды, несколько языков программирования, несколько типов приложений, — сколько получится сочетаний из *N* по 3? Чтобы ответить на этот вопрос и написана эта глава. Здесь, прежде чем погрузиться в детали программирования в Delphi 2005, мы разберемся, что и как можно разрабатывать в Delphi.

Многоязычная среда разработки

Основа знаний и опыта любого разработчика — языки программирования. В наше время изучение языков Basic, Pascal или C — это всего лишь начальный этап, первая ступенька к освоению инструментариев и технологий программирования. Поэтому разговор о возможностях Delphi 2005 мы начнем с языков программирования.

Некоторая универсальность и известное совершенство современных высокоуровневых языков программирования являются следствием долгого (по-

рядка полувека) развития начиная с первых, мало отличавшихся от наборов машинных кодов, средств взаимодействия человека и компьютера. При этом генеалогия некоторых языков весьма и весьма прихотлива. Например, язык программирования Delphi является дальним потомком небольшого семейства языков (Eiffel, Pascal), разработанных исключительно для обучения студентов программированию. Всего же человечество придумало больше тысячи различных языков программирования.

Столь прихотливая история развития и разнообразие языков для разработчиков объясняются двумя причинами. Первая причина созидательная — это стремление создать мощное и в то же время универсальное языковое средство, способное быстро и эффективно решать подавляющее большинство задач программирования. Вторая причина деструктивная — бурное развитие вычислительной техники, появление новых технологий программирования и направлений приложения усилий программистов. Ученые создавали очередной суперсовременный и универсальнейший язык, но еще более современные архитектуры и технологии быстро вытесняли его на периферию.

Примерно такая же ситуация повторилась несколько лет назад, следствием чего и стали многоязычные возможности Delphi 2005. К началу третьего тысячелетия признанными "вождями народа" языков программирования были Visual Basic и C++ — около 90% разработчиков использовали именно их. Язык же программирования Delphi, выступающий в одной весовой категории с C++ и ничем ему не уступающий, увы, распространен не так широко.

Но появление новой операционной среды Microsoft .NET и, как следствие, развитие новых технологий программирования показало, что "короли" оказались не готовыми к новым веяниям времени. На сцене появился новый язык программирования — C#.

Именно стремление к универсальности, желание дать разработчику инструментарий для решения наибольшего числа задач заставило создателей Delphi 2005 обеспечить поддержку нескольких языков программирования. Для этого потребовалось создать и включить в состав среды разработки новые компиляторы для языков C# и Visual Basic.

Итак, при работе в Delphi 2005 разработчику доступны три языка программирования: Delphi, C# и Visual Basic. У каждого из них есть свои преимущества, недостатки и предпочтительные сферы применения. Рассмотрим их.

Язык программирования Delphi

Язык программирования Delphi исторически является основным для среды разработки Delphi, он и его ближайшие предки применялись в Delphi начиная с первой версии. Во избежание путаницы здесь следует подробнее обсудить само название языка. Официальное название основного языка про-

граммирования среды разработки Delphi 2005 — Delphi. Оно употребляется начиная с версии Delphi 6. До этого язык программирования в среде разработки Delphi назывался Object Pascal. И хотя при выходе Delphi 6 фирма Borland четко и ясно объявила об изменении названия языка, до сих пор в некоторых публикациях в периодических изданиях и даже книгах встречается старое название языка — Object Pascal.

Что же подвигло разработчиков из Borland на переименование?

Действительно, ближайшим предком (папой) современного языка Delphi является Object Pascal, а дедушкой его стал старый, добрый классический Pascal. От своих предков язык Delphi унаследовал базовые конструкции, строгую типизацию, четкий и приближенный к человеческому синтаксис.

Но название языка было изменено отнюдь не случайно.

Дело в том, что бурное развитие объектно-ориентированного программирования (ООП) и визуального компонентного программирования, которые стали одними из важных причин появления среды разработки Delphi, потребовало существенного развития и языковых средств. И на основе языка Pascal был разработан Object Pascal. Он поддерживал работу с объектами и соответствовал парадигме ООП.

Но технологии программирования продолжали развиваться. Повсеместное внедрение технологий, основанных на COM (Component Object Model, компонентная модель объектов) в очередной раз потребовало адаптации языков программирования к новым условиям. Язык Object Pascal был дополнен интерфейсами, объекты были заменены на классы, появилось множественное наследование и т. д. Основой языка стали не объекты, а классы и интерфейсы. Поэтому Object Pascal был переименован в Delphi.

Язык программирования Delphi может быть использован для разработки приложений на операционных платформах Win32 и .NET. На нем реализована библиотека визуальных компонентов Delphi (Visual Component Library, VCL). Поэтому разработчики, продолжающие работу под Win32, могут и дальше создавать приложения, используя отработанные навыки и знания.

На основе проектов на языке Delphi возможна компиляция и для платформы .NET. Это позволяет сравнительно легко перенести приложения для Win32 на новую платформу .NET, продолжая применять привычные компоненты VCL, знакомые процедуры и функции.

В то же время при помощи языка Delphi можно создавать и полноценные приложения для .NET. Для этого в состав языка включены специфические конструкции, понятия, типы данных, необходимые для работы в .NET. А вместо библиотеки компонентов VCL имеется набор элементов управления, созданных специально для .NET и соответствующих стандартам .NET.

Язык программирования C#

Язык программирования C# был разработан как средство программирования для операционной среды .NET.

Авторы языка — Скотт Вилтамот (Scott Wiltamuth) и Андерс Хейлсберг (Anders Hejlsberg).

Синтаксис и возможности C# оптимизированы для работы с интерпретируемым кодом, в языке присутствуют конструкции, пригодные к использованию только при программировании для .NET. Например, язык поддерживает управление памятью в соответствии со стандартами .NET и обеспечивает сборку мусора. В синтаксическом наборе языка появились понятия сборки, пространства имен и т. д.

C# разработан на базе языков C++ и Java. Основной синтаксис, структура языка и модель типов данных свидетельствуют об этом весьма красноречиво. Но довольно существенная часть наследственных черт языков семейства C из нового языка исчезла. В C# больше нет некоторых синтаксических элементов (например, оператора `::`) и возможностей (например, опережающего объявления переменных).

В то же время несомненно и влияние на новый язык семейства языков Pascal. Поэтому исходный код на языке C# гораздо больше приближен к человеческому языку и лучше пригоден для восприятия.

В среде разработки Delphi проекты на языке C# можно создавать только для операционной среды .NET.

Язык программирования Visual Basic

Язык программирования Visual Basic — заслуженный ветеран объектно-ориентированного и компонентного программирования, известный большинству программистов на планете Земля. Его популярность и широкое распространение объясняются простотой изучения и использования. Но его возможности сравнительно невелики.

Если представить создаваемые программистами приложения в виде пирамиды по критерию сложности, то основание пирамиды займут программы, написанные на Visual Basic. Чем дальше мы будем подниматься к вершине, тем меньше будет встречаться Visual Basic и тем чаще будут попадаться проекты на C++, C# и Delphi.

В сложных проектах Visual Basic также применяется, но для решения ограниченных, специализированных задач. Например, для написания частей распределенных приложений для Internet, Web-служб или "тонких" клиентов. Именно для таких целей в среде разработки Delphi 2005 можно созда-

вать ограниченные по возможностям приложения на Visual Basic, но при этом нельзя применять визуальное программирование.

Программные платформы

Начиная с Delphi 8 разработчики могут создавать приложения для двух программных платформ (операционных сред): Win32 и .NET.

Традиционная платформа Win32 обеспечивает выполнение приложений, откомпилированных в двоичный машинный код, жестко привязанный к архитектуре процессора. Это обеспечивает повышенное быстродействие и эффективность программ, особенно на стадии загрузки. Ведь уже готовому машинному коду достаточно выделить область памяти, загрузить его и обеспечить общее управление операционной системой на уровне процесса.

Недостаток Win32 — жесткая зависимость от программно-аппаратной платформы и принципиальная ограниченность средств управления программным кодом. Очевидно, что однажды откомпилированный машинный код программы с точки зрения операционной системы представляет собой черный ящик со стандартизованными каналами (интерфейсами) управления и обмена данными. Управлять таким "ящиком" можно только в целом: поместить в память, выделить для работы дополнительный участок памяти, разрешить обращение к драйверам, выгрузить из памяти.

Новая платформа .NET ориентирована на 64-разрядную архитектуру процессоров, обеспечивает выполнение приложений, интерпретированных в специальный, универсальный, промежуточный язык — Intermediate Language (IL). Среда .NET выполняет первоначальную загрузку интерпретированного кода приложения, компиляцию в машинный код и управление. Такая архитектура обеспечивает независимость программ от аппаратной части компьютера и гораздо более эффективное управление, чем Win32.

К недостаткам формально следует отнести большие затраты на загрузку и управление приложением. Однако на современном этапе развития компьютерной техники этот недостаток нивелируется достаточными вычислительными мощностями. Кроме того, задержка выполнения приложения реальна только на этапе первоначальной загрузки. После компиляции в машинный код быстродействие программ .NET практически идентично Win32.

Ранее мы уже говорили о том, что в Delphi 2005 можно разрабатывать проекты на нескольких языках программирования. Основными из них являются Delphi и C#. При этом не все языки программирования одинаково универсальны. К примеру, C# специально создан для .NET, а Delphi имеет языковые подмножества для поддержки обеих платформ. Возможности Delphi 2005 по применению языков программирования для различных программных платформ представлены в табл. 1.1.

Таблица 1.1. *Применимость языков программирования к программным платформам*

	Платформа Win32	Платформа .NET
Delphi	+	+
C#	–	+

Компоненты .NET и VCL

Помимо языков программирования и операционных сред программист, работающий в Delphi 2005, должен учитывать и используемые здесь наборы визуальных и невизуальных компонентов.

> В Delphi 2005 вы можете разрабатывать приложения VCL для платформы Win32 на языке программирования Delphi (табл. 1.2).

Для программной платформы Win32 доступна библиотека компонентов VCL. Она очень разнообразна и обладает существенно бóльшими возможностями, чем набор компонентов для .NET. И это естественно: VCL существует гораздо дольше и не так сильно связана ограничениями на совместимость, как библиотека компонентов .NET. С другой стороны, часть компонентов VCL в .NET просто не нужна, так как соответствующие функции реализованы в .NET на уровне платформы. Например, не менее трети компонентов VCL в Delphi 7 и Delphi 2005 (компоненты IntraWeb и др.) обеспечивают функции, полноценно представленные в .NET в виде приложений ASP.NET.

> В Delphi 2005 вы можете разрабатывать приложения VCL для платформы .NET на языке программирования Delphi (см. табл. 1.2).

Переработанная библиотека VCL для платформы .NET в Delphi 2005 получила название VCL.NET. Она позволяет задействовать весь наработанный ранее богатый инструментарий и опыт разработчиков. Благодаря VCL.NET перенос приложений VCL на платформу .NET требует минимальных усилий. Большинство стандартных компонентов VCL и компонентов для приложений баз данных есть и в VCL.NET. Однако многие применяемые ранее технологии для разработки Internet-приложений (например, WebBroker) исчезли. Их необходимо заменять компонентами IntraWeb.

> В Delphi 2005 вы можете разрабатывать приложения для платформы .NET на языках программирования Delphi и C# (см. табл. 1.2).

Для "чистых" приложений .NET имеется собственный набор компонентов. В первую очередь, это визуальные компоненты Windows Forms, специфицированные на уровне архитектуры .NET. Другие компоненты .NET обеспечивают создание меню, диалоги, доступ к данным и т. д.

Таблица 1.2. *Возможности использования наборов компонентов в приложениях Delphi 2005*

	Платформа Win32		**Платформа .NET**	
	VCL	.NET	VCL	.NET
Delphi	+	–	+ (VCL.NET)	+
C#	–	–	–	+

Какие приложения можно создавать в Delphi

И в завершение главы мы обратимся к видам приложений, которые можно разрабатывать в Delphi 2005. Надо сказать, что разобраться в том, какие программы можно писать в Delphi 2005, не так просто. Чтобы разъяснить читателю все открывающиеся перед ним возможности и попытаться структурировать их более или менее четко, потребовалась отдельная глава! Увы, это обратная сторона медали, на лицевой стороне которой выгравирован девиз "универсальность" (см. начало этой главы). В общем-то, сложная и довольно пестрая картина возможностей, предлагаемых разработчикам в Delphi 2005, — есть объективное отражение состояния современного программирования. Сейчас мы находимся на переломе: происходит переход на новые операционные системы, новую архитектуру выполнения приложений, новые языки программирования, новые технологии и приемы работы.

В этой ситуации надо радоваться, что у нас есть универсальная среда программирования, которая обеспечивает работу и для Win32, и для .NET, и переход с одной платформы на другой. А неизбежная сложность и многообразие возможностей является лишь отражением реальных процессов.

Давайте попробуем собрать полученные знания в целостную картину того, что и как можно разрабатывать в Delphi 2005. В табл. 1.3 сведены виды приложений, доступные для использования, и дана их взаимосвязь с программными платформами и наборами компонентов. А затем дано краткое описание видов приложений.

Таблица 1.3. *Возможности программных платформ Win32 и .NET по созданию различных видов приложений*

	Платформа Win32	**Платформа .NET**	
	Язык Delphi	Язык Delphi	Язык C#
Приложения VCL	+	+	–
Приложения VCL.NET	–	+	–
Приложения Windows Forms	–	+	+
Приложения ADO.NET	–	+	+
Приложения баз данных	+	–	–
Приложения ASP.NET	–	+	+
Приложения IntraWeb	+	–	–
Приложения WebBroker	+	–	–
Приложения WebSnap	+	–	–
Приложения COM	+	+	+
Приложения ECO	–	+	+
Консольные приложения	+	+	+

Под приложениями VCL, VCL.NET и Windows Forms подразумеваются лишь способы создания пользовательского интерфейса и базовой функциональности любых приложений Windows, которые зависят от выбранного набора компонентов. Их выделение в отдельные виды в табл. 1.3 связано с тем, что эти библиотеки являются краеугольными камнями построения приложений на обеих программных платформах. Перечислим виды приложений, которые можно создавать в Delphi 2005.

- **Приложения ADO.NET** — приложения баз данных, разрабатываются исключительно на платформе .NET и обеспечивают доступ к данным только на основе технологии ADO (ActiveX Data Objects), включенной в состав архитектуры .NET. Декларируется, что ADO является основным способом доступа к данным в .NET, и именно поэтому такие приложения выделены в группу, отдельную от приложений баз данных.
- **Другие приложения баз данных** — объединяют все возможные в Delphi способы доступа к данным, за исключением ADO.NET. Например, доступ к данным через технологии dbGo, dbExpress и т. д. в Win32. Здесь необходимо отметить, что технология доступа к данным реализуется через наборы невизуальных компонентов. В связи с этим нужно понимать, что доступ к данным на платформе .NET при помощи технологии ADO

реализуется через ADO.NET, а такой же доступ к данным при помощи той же технологии ADO на платформе Win32 — через dbGo. Просто это разные наборы компонентов, которые созданы каждый для своей программной платформы, но используют один и тот же механизм доступа к данным ADO, заложенный в операционной системе.

- **Приложения ASP.NET** — специализированный вид приложений для Internet, которые можно создавать исключительно на платформе .NET.
- **Приложения IntraWeb, WebBroker и WebSnap** — объединены тем, что все они обеспечивают разработку Internet-приложений (Web-приложений и Web-служб) на программной платформе Win32. Их функциональные возможности значительно перекрываются технологией ASP.NET.
- **Приложения COM** — создаются в виде сервера или объекта COM, используют интерфейсы COM и при необходимости интерфейсы других технологий, дочерних COM. В терминологии .NET приложения COM называются приложениями с неуправляемым кодом, так как возможность их применения предусмотрена в архитектуре .NET, а реализованы эти приложения могут быть и на программной платформе Win32.
- **Приложения ECO** (Enterprise Core Objects) — создаются на основе возможностей архитектуры MDA (Model Driven Architecture). Такие приложения создаются с помощью модели на языке UML, на базе которой автоматически генерируется весь необходимый исходный код приложения.
- **Консольные приложения** — в Delphi 2005 можно разрабатывать для любой программной платформы и любого языка программирования. Эти приложения выполняются и управляются из командной строки без использования привычного визуального пользовательского интерфейса.

Перенос существующих приложений на платформу .NET

Обсуждая возможности, предоставляемые программистам средой разработки Delphi 2005, представляется необходимым разобраться и с проблемой переноса на новую платформу уже разработанных приложений. С языком программирования все в целом понятно — Delphi не требует бросить все и начать изучение C#. Язык программирования кардинальных изменений не претерпел, появились новые сущности, необходимые для работы с .NET, но это не влияет на уже существующий код для Win32.

Если вы попытаетесь открыть в Delphi 2005 любое приложение VCL, разработанное ранее, среда разработки предложит вам конвертировать приложение в один из двух возможных типов. Для этого используется диалог, представленный на рис. 1.1.

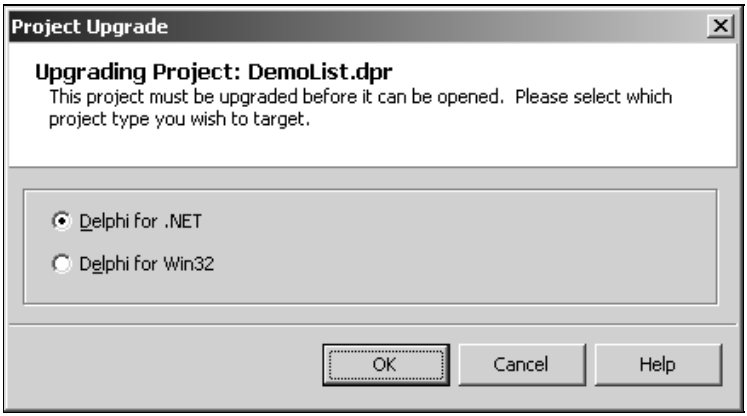


Рис. 1.1. Диалог конвертации приложения

Как видно из рис. 1.1, можно попробовать преобразовать старое приложение в приложение VCL для Win32 или для .NET. Если преобразование в Win32 не вызывает проблем, то с .NET ситуация гораздо сложней. В первую очередь, не совпадают наборы используемых компонентов. Основные компоненты (кроме специализированных компонентов для Web-приложений и Web-служб) реализованы и в VCL.NET, но конвертировать старое приложение в приложение Windows Forms для .NET скорее всего не удастся.

Кроме этого, нужно будет также убрать из кода переменные, типы данных которых напрямую адресуют память, так как в .NET такой код считается небезопасным. Есть и еще некоторые особенности, которые подробнее рассматриваются *в главе 3*. И конечно, в секциях `uses` нужно будет изменить имена модулей на соответствующие пространства имен.

Что же касается Internet-приложений, то здесь придется потрудиться. Многие компоненты, которые использовались в таких приложениях в Win32-версиях Delphi, удалены из палитры компонентов. Для приложений VCL.NET нужно выбирать компоненты IntraWeb, а для "чистой" платформы .NET — начинать разработку заново на ASP.NET.

При переносе приложений VCL с платформы Win32 на Windows Forms .NET неизбежны серьезные затруднения. Это касается не только компонентов VCL (в Windows Forms .NET их просто нет) и типов данных, но и собственно кода и бизнес-логики приложения. Привычные приемы работы со списками, файловой системой, памятью в .NET стали непригодными. Вы даже не сможете вывести на экран окно сообщения с сакраментальной фразой "Hello World!", так как в .NET процедуры `ShowMessage` просто нет. Поэтому приложения Windows Forms для .NET лучше всего начинать писать заново.

Резюме

В этой главе кратко рассказывается о возможностях разработки приложений в Delphi 2005. Их большое разнообразие объясняется тем, что Delphi 2005 выпущена в сложный период продвижения новой архитектуры .NET, когда необходимо поддерживать разработку как традиционных приложений, так и новых, а также обеспечивать миграцию приложений с платформы на платформу.

Многоязычность среды позволяет выполнять разработки на языках программирования Delphi, C# и Visual Basic.

В Delphi 2005 поддерживаются две программные платформы (операционные среды): .NET и Win32. Это позволяет создавать множество самых разных приложений.

Глава 2

Типы данных в Delphi

В этой главе рассматриваются основные типы данных, применяемые в Delphi, а также некоторые часто используемые конструкции языка. Знание основных типов данных облегчает программирование и позволяет избежать ряда ошибок начинающим программистам.

Все существующие в Delphi типы данных можно разделить на две основные группы. К первой относятся стандартные типы, предопределенные в среде разработки. На их основе разработчик может описывать собственные (определяемые) типы данных, которые относятся ко второй группе.

Определяемые типы данных существенно расширяют возможности программистов. Достаточно открыть любой файл с исходным кодом VCL, чтобы убедиться, что этот инструмент широко применяется разработчиками фирмы Borland. Однако при определении собственных типов данных требуется соблюдать осторожность и тщательно продумывать, какой предопределенный тип данных положить в основу собственного. Для этого необходимо иметь максимум информации о стандартных типах данных.

В главе рассматриваются следующие вопросы.

- Типы данных, используемые в Delphi.
- Тип вариант.
- Применение записей.
- Работа с массивами.

Материал главы рассчитан на читателя, имеющего представление о самом языке Delphi, его операторах и основных возможностях.

Простые типы

В эту категорию входят традиционные типы, большинство из которых сопровождают разработчиков чуть ли не с начала компьютерной эры. Сюда входят числовые, логический и символьный типы данных.

Несмотря на привычность и ординарность, в современной реализации простых типов данных имеется ряд особенностей, которые разработчик должен знать. В табл. 2.1 представлено описание простых типов, предназначенных для хранения чисел.

Таблица 2.1. *Простые числовые типы данных*

Тип данных	Значения	Число байтов в значении
`Integer`	−2147483648..2147483647	4 (со знаком)
`Cardinal`	0..4294967295	4
`ShortInt`	−128..127	1 (со знаком)
`SmallInt`	−32768..32767	2 (со знаком)
`LongInt`	−2147483648..2147483647	4 (со знаком)
`Int64`	$-2^{63}..2^{63}-1$	8 (со знаком)
`Byte`	0..255	1
`Word`	0..65535	2
`LongWord`	0..4294967295	4
`Real`	5.0×10^{-324} .. 1.7×10^{308}	8 (со знаком)
`Real48`	2.9×10^{-39} .. 1.7×10^{38}	6 (со знаком)
`Single`	1.5×10^{-45} .. 3.4×10^{38}	4 (со знаком)
`Double`	5.0×10^{-324} .. 1.7×10^{308}	8 (со знаком)
`Extended`	3.6×10^{-4951} .. 1.1×10^{4932}	10 (со знаком)
`Comp`	$-2^{63}+1$ .. $2^{63}-1$	8 (со знаком)
`Currency`	−922337203685477.5808.. 922337203685477.5807	8 (со знаком)

Для хранения целочисленных значений рекомендуется использовать базовые типы `Integer` (для положительных и отрицательных чисел) и `Cardinal` (для положительных чисел, вдвое больших `Integer`), так как они оптимизированы для выполнения вычислительных операций процессором.

Тип данных с плавающей точкой в языке Pascal появился, еще когда наличие сопроцессора (скажем, 80287) было скорее исключением, чем правилом. Все операции выполнялись в режиме эмуляции, для этого был выбран 6-байтовый (48-разрядный) формат, названный `Real`.

Сейчас процессоров без сопроцессора не бывает. Они работают со стандартизованными Американским институтом инженеров электротехники и электроники (IEEE) форматами: 80-разрядным (в Delphi — `Extended`), 64-разрядным (`Double`) и 32-разрядным (`Single`). Соответственно, старый добрый `Real` стал настоящим рудиментом, пожирающим массу времени на преобразование чисел.

На самом деле в Delphi под именем `Real` скрывается тип `Double`, что позволяет намного ускорить вычисления. Поэтому здесь необходимо учитывать следующие тонкости. Во-первых, размер всех массивов и структур с `Real` увеличится на 25%, что в среде Win32 не так уж страшно. Во-вторых, не работает код, в явном виде ориентированный на длину `Real` (6 байт). В этом случае можно описать переменные с новым типом `Real48`:

```
var r:real; r48:real48;
...
  ShowMessage(IntToStr(SizeOf(r)));
  ShowMessage(IntToStr(SizeOf(r48)));
...
```

В первом случае будет выдан результат 8, во втором — прежние 6.

Можно использовать директиву компилятора `{$REALCOMPATIBILITY ON}`, которая отменяет замену `Real` на `Double`. В этом случае описанный выше пример два раза выдаст ответ 6.

Для символьных данных (значение переменной данного типа хранит один символ) базовым является тип `Char`. На его основе созданы два дополнительных типа — `AnsiChar` и `WideChar`. Они обеспечивают работу с двумя кодировками символов: ANSI и Unicode.

Примечание

Кодировка ANSI отводит для хранения символа один байт. Кодировка Unicode является более сложной. В ней символы могут быть представлены одним или двумя байтами.

В Delphi расширены возможности логического типа данных. Помимо базового типа `Boolean` существуют дополнительные типы `ByteBool` (один байт), `WordBool` (два байта), `LongBool` (четыре байта), которые необходимы для обеспечения совместимости с функциями Windows и другими средами разработки (C++).

В базовом логическом типе все осталось по-прежнему: его возможные значения `True` (1) и `False` (0). В остальных типах значению `False` соответствует 0, всем остальным числам соответствует `True`.

Перечисляемый тип не претерпел никаких изменений. Он представляет собой упорядоченное множество значений:

```
type
    TSomeEnumeration = (seFirst, seSecond, seThrid);
var Enum1: TSomeEnumeration;
    Enum2: (Val1, Val2, Val3, Val4);
...
Enum1 := seSecond;
Enum2 := Val1;
...
```

Последний простой тип — поддиапазон — представляет собой подмножество любого другого базового простого типа:

```
type TOtherEnum = (Win95, NT4, Win98, Win2000, XP, Win 2003, LongHorn);
     TSub = 0..127;
var Sub1: TSub;
    Sub2: Win98..Win2000;
    Sub3: 'a'..'z';
...
Sub1 := 100;
Sub2 := NT4;
Sub3 := 'g';
...
```

Строковый тип

Самые разнообразные тексты встречаются в любых приложениях. Для их хранения можно применять константы и переменные. Для представления текстовых значений в Delphi есть три типа.

- Тип `ShortString` — может представлять текст длиной до 255 символов. Переменная этого типа имеет фиксированный размер 256 байтов. Первый байт отводится для хранения длины текста в символах. Остальные байты отведены для хранения символов.
- `AnsiString` — предназначен для создания переменных для хранения текстов произвольной длины. Тексты состоят из символов в кодировке ANSI (на один символ отводится один байт). Память для переменных этого типа выделяется динамически и ограничена только доступным адресным пространством. Сама переменная представляет собой указатель размером 4 байта. Если переменная пуста, указатель равен `Nil`. Две переменные,

имеющие одинаковые значения, ссылаются на одну область памяти (их указатели равны).

- Тип `WideString` — также осуществляет динамическое выделение памяти для хранения текста. Текст должен содержать символы в кодировке Unicode (два байта на символ).

Тип `ShortString` оставлен для обеспечения обратной совместимости с более ранними версиями. По умолчанию в проекте устанавливается директива компилятора `{$H+}`, которая определяет, что объявление

```
var SomeStr: String;
```

соответствует типу `AnsiString`. Директива `{$H-}` определяет, что такое объявление соответствует типу `ShortString`. Впрочем, можно употреблять типы `ShortString` и `AnsiString` напрямую.

Длину текста для строковой переменной можно ограничить числом символов, меньшим 255:

```
var SomeString[64];
```

К отдельному символу текста можно обратиться по его порядковому номеру (индексу):

```
var S: Char;
...
S := SomeString[4];
```

Помните, что для типа `ShortString` нулевой индекс соответствует одному байту длины текста.

Структурные типы

Структурные типы отличаются от простых тем, что переменная такого типа может содержать несколько значений простого типа одновременно. В зависимости от конкретного структурного типа это могут быть значения одного простого типа (множество, массив) или нескольких (запись).

Все структурные типы, кроме набора, могут объединять значения не только простых, но и структурных типов.

Данные в переменной структурного типа по умолчанию выравниваются по границе слова или двойного слова. Это сделано для ускорения доступа к данным.

Если переменная занимает много места, то быстротой можно пожертвовать ради экономии места — значения можно хранить упакованными. Для этого предназначено ключевое слово `packed`:

```
type TSomeStruct = packed set of 1..100;
```

Множества

Множество представляет собой набор значений одного из простых типов (кроме вещественного). Переменную-множество можно объявить через тип или напрямую:

```
type TSomeSet = set of Byte;
     TNumSet = 0..100;
var SetVar1: TSomeSet;
    SetVar2: set of 'A'..'Z';
```

В программе множеству можно присваивать произвольное подмножество:

```
SetVar1 := [1, 3, 5, 7, 11];
SetVar2 := ['X', 'Y', 'Z'];
```

Для контроля значений множеств имеется специальный оператор `in`:

```
if SetVar1 in [1, 3, 5]
  then ShowMessage('Простое число');
```

Массивы

В распоряжении программистов в Delphi есть массивы нескольких видов. Массивы могут быть открытыми, их размер теперь можно задавать динамически, также доступны многомерные динамические массивы. Однако обо всем по порядку.

Статические массивы

В Delphi нельзя при описании массива опустить ни нижнюю, ни верхнюю границы. Как правило, нумерация массивов начинается с нуля. Но если уж очень хочется начать нумерацию с единицы, теперь нужно написать:

```
var A: array[1..5] of integer;
```

При передаче массивов в качестве параметров процедур нельзя явно указывать границы. Следующее объявление функции

```
function atest(x: array[1..5] of integer): integer;
```

будет отвергнуто компилятором. Допускается только "безграничный" вариант:

```
function atest(x: array of integer): integer;
```

В Delphi существует строгая типизация. Так, в случае

```
Var
  A1: array[1..5] of integer;
  A2: array[1..5] of integer;
```

переменные A1 и A2 будут иметь разный тип! Естественно, массив A1 и формальный параметр функции Atest тоже будут иметь разный тип. Чтобы не нарушать канонов языка, массивы в функции передаются как динамические, т. е. вместе с адресом массива передается и число элементов в нем. Внутри тела функции можно получить значения граничных индексов с помощью функций Low() и High():

```
function atest(x: array of integer): integer;
var i: Integer;
begin
  Result := 0;
  for i:=Low(x) to High(x) do
    Result := Result + x[i];
end;
```

Есть и другой способ не отступить от канонов и остаться в рамках статических массивов — типизировать их:

```
type
 TMyArray = array[1..5] of integer;
Var
  A1: TMyArray;
function atest2(x: TMyArray): integer;
```

В этом случае компилятор не передает вызываемой функции длину массива — она ведь известна заранее. Это очень важно, если вы вызываете из программ на Delphi функции, написанные на других языках, — C, Fortran и т. д. Там "лишнее" двойное слово, содержащее длину массива, нарушит правильность работы. Поэтому присваивайте всем передаваемым массивам определенный тип.

Примечание

Обычно массивы передаются по ссылке, поэтому перед описанием таких параметров в функции следует указать ключевое слово var. Передача массивов, особенно больших, по значению (т. е. без ключевого слова var) может создать проблемы с памятью. Если вы боитесь употреблять тип var, а значения элементов не должны изменяться, опишите этот параметр с ключевым словом const.

Кстати, нумерация элементов массива не с 0 настолько несовременна, что может привести к курьезной ситуации. Передадим массив типа TMyArray в функцию, рассчитанную на динамический массив, и покажем значения границ:

```
function atest(x: array of integer): integer;
begin
```

```
  ShowMessage(IntToStr(Low(x)));
  ShowMessage(IntToStr(High(x)));
end;
```

Хотя границы равны 1 и 5, результатом вызова `Low` и `High` будут числа 0 и 4. Это и понятно, ведь была сообщена только длина!

Динамические массивы

В Delphi длина динамических массивов определяется уже во время выполнения программы. Описав переменную как

```
A1: array of integer;
```

мы имеем указатель, который будет соответствовать массиву вещественных чисел. Память под него выделяется процедурой `SetLength`:

```
SetLength(A1, 5);
```

В динамических массивах нумерация начинается только с 0. Хотя это по природе своей указатели, знак разыменования (^) здесь, как и в отношении объектов, не применяется:

```
A[0] := A[5].
```

То обстоятельство, что динамические массивы являются указателями, приводит к определенным нюансам в их использовании. Первый из них касается правил присвоения массивов. Запись:

```
Var A1,A2 : array[0..4] of integer;
...
A2 := A1;
...
```

означает, что всем элементам массива `A2` присваиваются значения соответствующих элементов из массива `A1`; но при этом `A1` и `A2` — разные переменные, хранящиеся в разных областях памяти.

Не так обстоит дело с динамическими массивами. Присвоение их друг другу — не более чем присвоение указателей. В случае

```
Var A1,A2 : array of real;
...
SetLength(A1,5);
A1[0] := 0.0;
A2 := A1;
A2[0] := 2.0;
```

значение `A1[0]` также изменится и будет равно 2. Это произошло потому, что после присвоения указатели `A1` и `A2` ссылаются на один и тот же адрес в

памяти, и изменения элементов одного массива приведут к синхронным изменениям в другом.

Освободить выделенную память можно, вызвав процедуру `finalize`:

```
Finalize(A1);
```

или просто присвоив указателю `A1` значение `nil`. В этом случае память не потеряется (как могло бы показаться на первый взгляд), а будет вызвана соответствующая процедура, которая корректно освободит память. Причем память будет освобождаться только тогда, когда на нее больше не ссылается ни одна динамически созданная структура. В рассмотренном только что примере с массивами `A1` и `A2` уничтожение `A1` не означает, что тут же пропадет и `A2` — нет, он будет указывать все на тот же участок памяти, который первоначально выделялся под `A1`.

Дальнейшим развитием идеи динамических массивов являются многомерные динамические массивы. Внешне все выглядит так же (описание без указания границ):

```
Var AA: array of array of integer;
```

В функции `SetLength` после имени переменной нужного типа можно задать сразу все размерности массива:

```
SetLength(AA, 10, 5);
```

и дальше работать с элементами массива:

```
...
AA[9,4] := 1;
AA[9][4] := 1;
...
```

Если не установлена опция компилятора `{$R+}` (проверка на выход за границы массива), будьте осторожны с индексами. В этом случае присвоение значения элементу `AA[4,9]` не будет воспринято как ошибка, и вместо него будут фактически испорчены сразу два элемента: `AA[5,2]` и `AA[5,3]` (два, а не один — из-за нечетной длины строки). В памяти массивы расположены так, что быстрее всего изменяется последний индекс, затем предпоследний и т. п.

Многомерные динамические массивы

Можно пойти еще дальше и создать динамические массивы с переменной длиной по разным индексам, т. е. первая строка будет состоять из n1 элементов, вторая — из n2 и т. д. Интерес к таким массивам, называемым еще разреженными, не праздный — и в вычислительной математике, при индексировании баз данных, и в других областях бывает необходимо работать с

такой большой размерностью массивов, которая полностью исключает их размещение в памяти. В то же время известно, что большинство элементов массива — нули; тогда мы отводим под каждую строку ровно столько памяти, сколько требуется, и решаем эту проблему.

Для создания такого массива сначала нужно задать его размерность по первому индексу:

```
Var AA: array of array of integer;
...
SetLength(AA, 10);
```

Это означает, что массив будет состоять из 10 строк. Теперь длина каждой задается отдельно:

```
...
SetLength(AA[0], 5);
SetLength(AA[1], 6);
SetLength(AA[2], 7);
...
```

По-прежнему можно освободить память одним-единственным присвоением указателю `AA` значения `nil` или вызовом функции `Finalize`. "Умная" библиотека времени выполнения освободит память каждой строки отдельно.

В использовании многомерных динамических массивов есть одно "но". Передавать их как параметры в функции нельзя. Не допускается конструкция вида:

```
...
function atest(x: array of array of integer): integer;
...
```

В этом случае программисту часть работы придется сделать самому, учитывая сказанное выше. Если передать в процедуру переменную типа `array of real`, то она получит в качестве параметра указатель на массив указателей, которые в свою очередь будут указывать на данные.

Записи

Запись представляет собой специально объявленную структуру, объединяющую переменные различных типов. Каждая переменная записи называется полем. Именно запись послужила основой для появления структуры объекта, а затем класса *(см. главу 3)*. Запись объявляется так:

```
type
TPoint = record
```

```
  x: LongInt;
  y: LongInt;
end;
```

Здесь приведено объявление записи, обеспечивающей в Delphi хранение координат точки.

В записи в качестве полей допустимы любые типы, в том числе и записи:

```
type
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

В Delphi эта запись служит для хранения данных о координатах прямоугольника.

Записи можно объявлять сразу как переменные:

```
var
 Record1: record
           Field1: Integer;
           Field2: Double;
          end;
```

Обращение к полям записи осуществляется так:

```
var Point1: TPoint;
    Rect1: TRect;
...
 Point1.X := 10;
 Point1.Y := 20;
 Rect1.Left := 5;
 Rect1.Top := 5;
 Rect1.BottomRight := Point1;
...
```

Как видно из примера, присваивание возможно не только по отношению к полям, но и между записями в целом.

Варианты

В практике программирования встречаются ситуации, когда передаваемое переменной значение может иметь неизвестный заранее тип данных. Ино-

гда в такой ситуации можно воспользоваться приведением значений к строковому типу. Однако это не всегда возможно. В таких ситуациях удобен специальный тип — вариант.

Для представления значения в варианте отводится 16 байтов. В них содержится код типа и само значение или указатель на значение.

Вариант позволяет хранить данные любых типов, за исключением структурных типов и типа `Int64`. Допускается хранение динамических массивов (в вариант помещается указатель на область памяти массива).

При инициализации переменной-варианта ей присваивается значение `Unassigned`.

Получить информацию о типе хранимых данных можно с помощью специальной функции `VarType`:

```
var VarValue: Variant;
    Str1: String;
...
  if VarType(VarValue) = varString
    then Str1 := VarValue;
```

В табл. 2.2 приведены коды типов варианта.

Таблица 2.2. *Коды типов варианта*

Код типа	Значение	Описание
`varEmpty`	`$0000`	Вариант не задан после создания (значение `Unassigned`)
`varNull`	`$0001`	Вариант не имеет значения
`varSmallint`	`$0002`	Значение варианта имеет тип `SmallInt`
`varInteger`	`$0003`	Значение варианта имеет тип `Integer`
`varSingle`	`$0004`	Значение варианта имеет тип `Single`
`varDouble`	`$0005`	Значение варианта имеет тип `Double`
`varCurrency`	`$0006`	Значение варианта имеет тип `Currency`
`varDate`	`$0007`	Вариант хранит дату и время
`varOleStr`	`$0008`	Значение варианта имеет тип `WideString`
`varDispatch`	`$0009`	Вариант хранит указатель на объект Автоматизации (интерфейс `IDispatch`)
`varError`	`$000A`	Возвращает код системной ошибки
`varBoolean`	`$000B`	Значение варианта имеет логический тип

Таблица 2.2 *(окончание)*

Код типа	Значение	Описание
varUnknown	$000D	Вариант хранит указатель на объект COM (интерфейс IUnknown)
varByte	$0011	Значение варианта имеет тип Byte
varString	$0100	Значение варианта имеет тип AnsiString
varTypeMask	$0FFF	Возвращает битовую маску типа
varArray	$2000	Вариант хранит массив
varByRef	$4000	Вариант хранит ссылку

При необходимости в вариант можно передавать и обычные массивы. Для этого применяются функции VarArrayCreate и VarArrayOf:

```
function VarArrayCreate(const Bounds: array of Integer; VarType:
Integer): Variant;
function VarArrayOf(const Values: array of Variant): Variant;
```

Вот пример их использования:

```
var VarValue: Variant;
...
VarValue := VarArrayCreate([1, 2, 3, 4], varInteger);
VarValue := VarArrayOf([True, 10, 'SomeString', 2.8954]);
```

Резюме

Строгая типизация языка Delphi обусловливает необходимость знания доступных в Delphi типов данных. Умение использовать дополнительные возможности языка, такие как записи и динамические и многомерные массивы, — это еще один шаг на пути к разработке профессиональных приложений.

Глава 3

Объектно-ориентированное программирование

Изучение основ объектно-ориентированного программирования (ООП) входит в минимум знаний, необходимых программисту. Но от понимания объектов до их грамотного применения — дистанция огромного размера. Для создания более или менее сложных приложений нужны навыки объектно-ориентированного дизайна, а для этого, в свою очередь, необходимо четкое знание возможностей вашей среды программирования. Поэтому в этой главе мы постараемся акцентировать внимание читателя на применении ООП в среде Delphi.

По сравнению с традиционными способами программирования ООП обладает рядом преимуществ. Главное из них заключается в том, что эта концепция в наибольшей степени соответствует внутренней логике функционирования операционной системы Windows. Программа, состоящая из отдельных объектов, отлично приспособлена к реагированию на события, происходящие в ОС. К другим преимуществам ООП можно отнести большую надежность кода и возможность повторного использования отработанных объектов.

В этой главе рассматриваются способы реализации основных механизмов ООП в Delphi.

- Понятия объекта, класса и компонента.
- Основные механизмы ООП: инкапсуляция, наследование и полиморфизм.
- Особенности реализации объектов.
- Взаимодействие свойств и методов.

Материал главы рассчитан на читателя, имеющего представление о самом языке Delphi, его операторах и основных возможностях.

Объект и класс

Перед началом работы необходимо ввести основные понятия и определения.

Классом в языке Delphi называется структура языка, которая может иметь в своем составе переменные, функции и процедуры. Переменные в зависимости от предназначения называются *полями* или *свойствами*. Процедуры и функции класса называются *методами*. Соответствующий классу тип будем называть *объектным типом:*

```
type
  TMyObject = class(TObject)
    MyField: Integer;
    function MyMethod: Integer;
end;
```

В этом примере описан класс `TMyObject`, содержащий поле `MyField` и метод `MyMethod`.

Поля объекта аналогичны полям записи (`record`). Это — данные, уникальные для каждого созданного в программе экземпляра класса. Описанный здесь класс `TMyObject` имеет одно поле — `MyField`.

Методы — это процедуры и функции, описанные внутри класса и предназначенные для операций над его полями. В состав класса входит указатель на специальную таблицу, где содержится вся информация, нужная для вызова методов. От обычных процедур и функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут поля именно того объекта, который вызвал метод. Внутри метода указатель на вызвавший его объект доступен под зарезервированным именем `Self`.

Понятие свойства будет подробно рассмотрено далее. Пока можно определить его как поле, доступное для чтения и записи не напрямую, а через соответствующие методы.

Классы могут быть описаны либо в секции интерфейса модуля, либо на верхнем уровне вложенности секции реализации. Не допускается описание классов "где попало", т. е. внутри процедур и других блоков кода.

Разрешено опережающее объявление классов, как в следующем примере:

```
type
  TFirstObject = class;
  TSecondObject = class(TObject)
  F1st : TFirstObject;
  ...
  end;
```

```
  TFirstObject = class(TObject)
  ...
  end;
```

Чтобы использовать класс в программе, нужно, как минимум, объявить переменную этого типа. Переменная объектного типа называется *экземпляром класса*, или *объектом*:

```
var
  AMyObject: TMyObject;
```

Как создаются и уничтожаются объекты?

Те, кто раньше использовал ООП в работе на C++, будьте внимательны: в Delphi экземпляры объектов могут быть только динамическими. Это означает, что в приведенном выше фрагменте кода переменная `AMyObject` на самом деле является указателем, содержащим адрес объекта.

Объект "появляется на свет" в результате вызова специального метода, который инициализирует объект — *конструктора*. Созданный экземпляр уничтожается другим методом — *деструктором*:

```
AMyObject := TMyObject.Create;
{ действия с созданным объектом }
...
AMyObject.Destroy;
```

"Но, — скажет внимательный читатель,— ведь объекта еще нет, как же мы можем вызывать его методы?" Справедливое замечание. Однако обратите внимание, что вызывается метод `TMyObject.Create`, а не `AMyObject.Create`. Есть такие методы (в том числе конструктор), которые успешно работают до (или даже *без*) создания объекта. О таких методах, называемых *методами класса*, пойдет речь далее.

В Delphi конструкторов у класса может быть несколько. Общепринято называть конструктор `Create` (в отличие от C++, где его имя совпадает с именем класса). Типичное название деструктора — `Destroy` (листинг 3.1).

Листинг 3.1. Пример описания объекта

```
type
  TMyObject = class(TObject)
    MyField: Integer;
    Constructor Create;
    Destructor Destroy;
    Function MyMethod: Integer;
end;
```

Для уничтожения экземпляра объекта рекомендуется метод `Free`, который первоначально проверяет указатель (не равен ли он `Nil`), и только затем вызывает `Destroy`:

```
AMyObject.Free;
```

До передачи управления телу конструктора происходит собственно создание объекта — под него отводится память, значения всех полей обнуляются. Далее выполняется код конструктора, написанный программистом для инициализации экземпляров данного класса. Таким образом, хотя на первый взгляд синтаксис конструктора схож с вызовом процедуры (не определено возвращаемое значение), но на самом деле конструктор — это функция, возвращающая созданный и инициализированный объект.

Примечание

Конструктор создает новый объект только в том случае, если перед его именем указано имя класса. Если указать имя уже существующего объекта, он поведет себя по-другому: не создаст новый объект, а только выполнит код, содержащийся в теле конструктора.

Чтобы правильно инициализировать в создаваемом объекте поля, относящиеся к классу-предку, нужно сразу же при входе в конструктор вызвать конструктор предка при помощи зарезервированного слова `inherited`:

```
constructor TMyObject.Create;
begin
 inherited Create;
...
end;
```

Взяв любой из примеров, прилагаемых к этой книге или поставляемых вместе в Delphi, вы почти не увидите там вызовов конструкторов и деструкторов. Дело в том, что каждый компонент, попавший при визуальном проектировании в ваше приложение из Палитры инструментов, включается в определенную иерархию. Иерархия эта замыкается на форме (класс `TForm`): для всех ее составных частей конструкторы и деструкторы вызываются автоматически, незримо для программиста. Кто создает и уничтожает формы? Это делает приложение (глобальный объект с именем `Application`). В файле проекта (с расширением dpr) вы можете увидеть вызовы метода `Application.CreateForm`, предназначенного для этой цели.

Что же касается объектов, создаваемых динамически (во время выполнения приложения), то здесь нужен явный вызов конструктора и метода `Free`.

Поля, свойства и методы

Поля класса являются переменными, объявленными внутри класса. Они предназначены для хранения данных во время работы экземпляра класса (объекта). Ограничений на тип полей в классе не предусмотрено. В описании класса поля должны предшествовать методам и свойствам. Обычно поля служат для обеспечения выполнения операций внутри класса.

Примечание

При объявлении имен полей принято к названию добавлять заглавную букву `F`. Например `FSomeField`.

Итак, поля предназначены для использования внутри класса. Однако класс должен каким-либо образом взаимодействовать с другими классами или программными элементами приложения. В подавляющем большинстве случаев класс должен выполнить с некоторыми данными определенные действия и представить результат.

Для получения и передачи данных в классе применяются свойства. Объявление свойств в классе осуществляется с помощью зарезервированного слова `property`.

Свойства представляют собой атрибуты, которые составляют индивидуальность объекта и помогают описать его. Например, обычная кнопка в окне приложения обладает такими свойствами, как цвет, размеры, положение. Для экземпляра класса "кнопка" значения этих атрибутов задаются при помощи свойств — специальных переменных, определяемых ключевым словом `property`. Цвет может задаваться свойством `Color`, размеры — свойствами `Width` и `Height` и т. д.

Так как свойство обеспечивает обмен данными с внешней средой, то для доступа к его значению существуют специальные методы класса. Поэтому обычно свойство определяется тремя элементами: полем и двумя методами, которые осуществляют его чтение/запись:

```
type
 TAnObject = class(TObject)
  function GetColor: TSomeType;
  procedure SetColor(ANewValue: TSomeType);
  property AColor: TSomeType read GetColor write SetColor;
 end;
```

В данном примере доступ к значению свойства `AColor` осуществляется через вызовы методов `GetColor` и `SetColor`.

Однако в обращении к этим методам в явном виде нет необходимости, достаточно написать

```
AnObject.AColor := AValue;
AVariable := AnObject.AColor;
```

и компилятор самостоятельно оттранслирует обращение к свойству в вызовы методов. То есть внешне свойство выглядит в точности как обычное поле, но за всяким обращением к нему могут стоять нужные вам действия. Например, если у вас есть объект, представляющий собой квадрат на экране, и его свойству "цвет" вы присваиваете значение "белый", то произойдет немедленная перерисовка, приводящая реальный цвет на экране в соответствие со значением свойства. Выполнение этой операции осуществляется методом, который связан с установкой значения свойства "цвет".

В методах, входящих в состав свойств, может осуществляться проверка устанавливаемой величины на попадание в допустимый диапазон значений и вызов других процедур, зависящих от вносимых изменений. Если же потребности в специальных процедурах чтения и/или записи нет, можно вместо имен методов применять имена полей. Рассмотрим следующую конструкцию (листинг 3.2).

Листинг 3.2. Пример обращения к свойствам

```
TPropObject = class(TObject)
    FValue: TSomeType;
    procedure DoSomething;
    function Correct(AValue: Integer):boolean;
    procedure SetValue(NewValue: Integer);
    property AValue: Integer read FValue write SetValue;
  end;
...
procedure TPropObject.SetValue(NewValue: Integer);
begin
  if (NewValue<>FValue) and Correct(NewValue)
    then FValue := NewValue;
  DoSomething;
end;
```

В этом примере обращение к свойству `AValue` означает просто чтение поля `FValue`. Зато при присвоении значения внутри `SetValue` вызываются сразу два метода.

Если свойство должно только читаться или только записываться, в его описании может присутствовать только соответствующий метод:

```
type
  TAnObject = class(TObject)
    property AProperty: TSomeType read GetValue;
  end;
```

В этом примере вне объекта значение свойства можно лишь прочитать, попытка присвоить свойству `AProperty` значение вызовет ошибку компиляции.

Для присвоения свойству значения по умолчанию используется ключевое слово `default`:

```
property Visible: boolean read FVisible write SetVisible default True;
```

Это означает, что при запуске программы свойство будет установлено компилятором в `True`.

Свойство может быть и векторным; в этом случае оно внешне выглядит как массив:

```
property APoints[Index : Integer]:TPoint read GetPoint write SetPoint;
```

На самом деле в классе может и не быть соответствующего поля — массива. Напомним, что вся обработка обращений к внутренним структурам класса может быть замаскирована.

Для векторного свойства необходимо описать не только тип элементов массива, но также имя и тип индекса. После ключевых слов `read` и `write` в этом случае должны стоять имена методов (использование здесь полей массивов недопустимо). Метод, читающий значение векторного свойства, должен быть описан как функция, возвращающая значение того же типа, что и элементы свойства, и имеющая единственный параметр того же типа и с тем же именем, что и индекс свойства:

```
function GetPoint(Index:Integer):TPoint;
```

Аналогично метод, помещающий значения в такое свойство, должен первым параметром иметь индекс, а вторым — переменную нужного типа (которая может быть передана как по ссылке, так и по значению):

```
procedure SetPoint(Index:Integer; NewPoint:TPoint);
```

У векторных свойств есть еще одна важная особенность. Некоторые классы в Delphi (списки `TList`, наборы строк `TStrings`) "построены" вокруг основного векторного свойства. Основной метод такого класса дает доступ к некоторому массиву, а все остальные методы являются как бы вспомогательными. Специально для облегчения работы в этом случае векторное свойство может быть описано с ключевым словом `default`:

```
type
 TMyObject = class;
```

```
property Strings[Index: Integer]: string read Get write Put; default;
end;
```

Если у объекта есть такое свойство, то можно его не упоминать, а ставить индекс в квадратных скобках сразу после имени объекта:

```
var AMyObject: TMyObject;
begin
...
AMyObject.Strings[1] := 'First'; {первый способ}
AMyObject[2] := 'Second'; {второй способ}
...
end.
```

Будьте внимательны, применяя зарезервированное слово `default`, — как мы увидели, для обычных и векторных свойств оно употребляется в разных случаях и с разным синтаксисом.

О роли свойств в Delphi красноречиво говорит следующий факт: у всех имеющихся в распоряжении программиста стандартных классов 100% полей недоступны и заменены базирующимися на них свойствами. Рекомендуем при разработке собственных классов придерживаться этого же правила.

Внимательный читатель обратил внимание, что при объяснении терминов "поле" и "свойство" мы упоминали понятие метода, и наверняка понял его общий смысл. Итак, методом называется объявленная в классе функция или процедура, которая используется для работы с полями и свойствами класса. Согласно принципам ООП *(см. разд. "Инкапсуляция" данной главы)*, обращаться к свойствам класса можно только через его методы. От обычных процедур и функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут данные именно того объекта, который вызвал метод. На некоторых особенностях работы с методами мы остановимся далее.

События

Программистам, давно работающим в Windows, наверное, не нужно пояснять смысл термина "событие". Сама эта среда и написанные для нее программы управляются событиями, возникающими в результате воздействий пользователя, аппаратуры компьютера или других программ. Весточка о наступлении события — это сообщение Windows, полученное так называемой функцией окна. Таких сообщений сотни и, по большому счету, написать программу для Windows — значит, определить и описать реакцию на некоторые из них.

Работать с таким количеством сообщений, даже имея под рукой справочник, нелегко. Поэтому одним из больших преимуществ Delphi является то,

что программист избавлен от подобной необходимости (хотя возможность этого у него есть). Типовых событий в Delphi — не более трех десятков, и все они имеют простую интерпретацию, не требующую глубоких знаний среды.

Рассмотрим, как реализованы события на уровне языка Delphi. Событие — это свойство процедурного типа, предназначенное для создания пользовательской реакции на то или иное входное воздействие:

```
property OnMyEvent: TMyEvent read FOnMyEvent write FOnMyEvent;
```

Здесь `FOnMyEvent` — поле процедурного типа, содержащее адрес некоторого метода. Присвоить такому свойству значение — значит, указать объекту адрес метода, который будет вызываться в момент наступления события. Такие методы называют обработчиками событий. Например, запись:

```
Application.OnActivate := MyActivatingMethod;
```

означает, что при активизации объекта `Application` (так называется объект, соответствующий работающему приложению) будет вызван метод-обработчик `MyActivatingMethod`.

Внутри библиотеки времени выполнения (RTL) Delphi вызовы обработчиков событий находятся в методах, обрабатывающих сообщения Windows. Выполнив необходимые действия, этот метод проверяет, известен ли адрес обработчика, и, если это так, вызывает его:

```
if Assigned(FOnMyEvent) then FOnMyEvent(Self);
```

События имеют разное количество и тип параметров в зависимости от происхождения и предназначения. Общим для всех является параметр `Sender` — он указывает на объект-источник события. Самый простой тип — `TNotifyEvent` — не имеет других параметров:

```
TNotifyEvent = procedure (Sender: TObject) of object;
```

Тип метода, предназначенный для извещения о нажатии клавиши, предусматривает передачу программисту кода этой клавиши, о передвижении мыши — ее текущих координат и т. п. Все события в Delphi принято предварять префиксом `On`: `OnCreate`, `OnMouseMove`, `OnPaint` и т. д. Дважды щелкнув в Инспекторе объектов на странице **Events** в поле любого события, вы получите в программе заготовку метода нужного типа. При этом его имя будет состоять из имени текущего компонента и имени события (без префикса `On`), а относиться он будет к текущей форме. Пусть, например, на форме `Form1` есть текст `Label1`. Тогда для обработки щелчка мышью (событие `OnClick`) будет создана заготовка метода `TForm1.Label1Click`:

```
procedure TForm1.Label1Click(Sender: TObject);
begin
end;
```

Поскольку события являются свойствами объекта, их значения можно изменять в любой момент во время выполнения программы. Эта замечательная возможность называется делегированием. Можно в любой момент взять способы реакции на события у одного объекта и присвоить (делегировать) их другому:

```
Object1.OnMouseMove := Object2.OnMouseMove;
```

Принцип делегирования позволяет избежать трудоемкого процесса порождения новых дочерних классов для каждого специфического случая, заменяя его простой подстановкой процедур. При необходимости можно выбирать один из нескольких возможных вариантов обработчиков событий.

Но какой механизм позволяет подменять обработчики, ведь это не просто процедуры, а методы? Здесь как нельзя кстати приходится существующее в Delphi понятие указателя на метод. Отличие метода от процедуры состоит в том, что помимо явно описанных параметров методу всегда неявно передается еще и указатель на вызвавший его экземпляр класса (переменная `Self`). Вы можете описать процедурный тип, который будет совместим по присваиванию с методом (т. е. предусматривать получение `Self`). Для этого в описание процедуры нужно добавить зарезервированные слова `of object`. Указатель на метод — это указатель на такую процедуру (листинг 3.3).

Листинг 3.3. Пример делегирования

```
type
 TMyEvent = procedure(Sender: TObject; var AValue: Integer) of object;

 T1stObject = class;
  FOnMyEvent: TMyEvent;
  property OnMyEvent: TMyEvent read FOnMyEvent write FOnMyEvent;
 end;
 T2ndObject = class;
  procedure SetValue1(Sender: TObject; var AValue: Integer);
  procedure SetValue2(Sender: TObject; var AValue: Integer);
 end;
...
var
  Obj1: T1stObject;
  Obj2: T2ndObject;
begin
 Obj1 := T1stObject.Create;
 Obj2 := T2ndObject.Create;
```

```
 Obj1.OnMyEvent := Obj2.SetValue1;
 Obj1.OnMyEvent := Obj2.SetValue2;
...
end.
```

Этот пример показывает, что при делегировании можно присваивать методы других классов. Здесь обработчиком события `OnMyEvent` объекта `Obj1` по очереди выступают методы `SetValue1` и `SetValue2` объекта `Obj2`.

Обработчики событий нельзя сделать просто процедурами — *они обязательно должны быть чьими-то методами*. Но их можно "отдать" какому-либо другому объекту. Более того, для этих целей можно описать и создать специальный объект. Его единственное предназначение — быть носителем методов, которые затем делегируются другим объектам. Разумеется, такой объект надо не забыть создать до использования его методов, а в конце — уничтожить. Другая возможность будет рассмотрена в одном из последующих разделов.

Мы сейчас решили задачу использования нескольких разных обработчиков того или иного события для одного объекта. Часто требуется решить обратную задачу — а как применить для разных событий различных объектов один и тот же обработчик?

Если никакой "персонификации" объекта, вызвавшего метод, не нужно, все делается тривиально и никакой проблемы не возникает. Самый простой пример: в современных программах основные функции дублируются дважды (в меню и в панели инструментов). Естественно, сначала нужно создать и наполнить метод содержимым (скажем, для пункта меню), а затем в Инспекторе объектов указать его же для кнопки панели инструментов.

Более сложен случай, когда внутри такого метода нужно разобраться, кто собственно его вызвал. Если потенциальные кандидаты имеют разный объектный тип (кнопка и пункт меню), то именно его можно применить в качестве критерия:

```
If Sender is TMenuItem then ShowMessage('Выбран пункт меню');
```

Если же все объекты, разделяющие между собой один обработчик события, относятся к одному классу, то приходится прибегать к дополнительным ухищрениям. Типовой прием — через свойство `Tag`, которое имеется у всех компонентов и, вполне вероятно, именно для этого и задумывалось:

```
const colors : array[0..7] of TColor =
(clWhite,clRed,clBlue,clYellow,clAqua,clGreen,clMaroon,clBlack);
procedure TForm1.CheckBoxClick(Sender: TObject);
begin
 with TCheckBox(Sender) do
```

```
  if Checked
   then Color := Colors[Tag]
   else Color := clBtnFace;
end;
```

Пусть на форме имеется несколько переключателей. Для того чтобы при нажатии каждый из них окрашивался в свой цвет, нужно в Инспекторе объектов присвоить свойству `Tag` значения от 0 до 7 и для каждого связать событие `OnClick` с методом `CheckBoxClick`. Этот единственный метод справится с задачей для всех переключателей.

Инкапсуляция

В предыдущих разделах мы ввели ряд новых понятий, которыми будем пользоваться в дальнейшем. Теперь поговорим о принципах, составляющих суть объектно-ориентированного программирования. Таких принципов три: инкапсуляция, наследование и полиморфизм.

Как правило, объект — это сложная конструкция, свойства и поведение составных частей которой находятся во взаимодействии. К примеру, если мы моделируем взлетающий самолет, то после набора им определенной скорости и отрыва от земли принципы управления им полностью изменяются. Поэтому в объекте "самолет" явно недостаточно просто увеличить значение поля "скорость" на несколько километров в час, такое изменение должно автоматически повлечь за собой целый ряд других.

При создании программных объектов подобные ситуации можно моделировать, связывая со свойствами необходимые методы. Понятие инкапсуляции соответствует этому механизму.

Классическое правило объектно-ориентированного программирования утверждает, что для обеспечения надежности нежелателен прямой доступ к полям объекта: чтение и обновление их содержимого должны производиться посредством вызова соответствующих методов. Это правило и называется инкапсуляцией. В старых реализациях ООП эта мысль внедрялась только посредством призывов и примеров в документации, в языке же Delphi есть соответствующая конструкция. В Delphi пользователь вашего объекта может быть полностью "отгорожен" от полей при помощи свойств.

Примечание

Дополнительные возможности ограничения доступа к нужным данным обеспечивают области видимости.

Наследование

Вторым "столпом" ООП является наследование. Этот простой принцип означает, что если вы хотите создать новый класс, лишь немного отличающийся от старого, то совершенно нет необходимости в переписывании заново уже существующих полей и методов. Вы объявляете, что новый класс `TNewObject`

```
TNewObject = class(TOldObject);
```

является *потомком* или *дочерним классом* старого класса `TOldObject`, называемого *предком* или *родительским классом*, и добавляете к нему новые поля, методы и свойства — иными словами, то, что нужно при переходе от общего к частному.

Примечание

Прекрасный пример, иллюстрирующий наследование, представляет собой иерархия классов VCL.

В Delphi все классы являются потомками класса `TObject`. Поэтому если вы создаете дочерний класс прямо от `TObject`, то в определении его можно не упоминать. Следующие два выражения одинаково верны:

```
TMyObject = class(TObject);
TMyObject = class;
```

Первый вариант, хотя он и более длинный, предпочтительнее — для устранения возможных неоднозначностей. Класс `TObject` несет очень серьезную нагрузку и будет рассмотрен отдельно.

Унаследованные от класса-предка поля и методы доступны в дочернем классе. Если имеет место совпадение имен методов, то говорят, что они перекрываются.

Поведение методов при наследовании, без преувеличения, является краеугольным камнем объектно-ориентированного программирования. В зависимости от того, какие действия происходят при вызове, методы делятся на три группы. В первую группу отнесем статические методы, во вторую — виртуальные (`virtual`) и динамические (`dynamic`) и, наконец, в третью — появившиеся только в Delphi 4 перегружаемые (`overload`) методы.

Методы первой группы полностью перекрываются в классах-потомках при их переопределении. При этом можно полностью изменить объявление метода. Методы второй группы при наследовании должны сохранять наименование и тип. Перегружаемые методы дополняют механизм наследования возможностью выбрать нужный вариант метода (собственный или родительский) в зависимости от условий применения. Подробно все эти методы обсуждаются далее.

Язык C++ допускает так называемое множественное наследование. В этом случае новый класс может наследовать часть своих элементов от одного родительского класса, а часть — от другого. Это, наряду с удобствами, зачастую приводит к проблемам. В Delphi понятие множественного наследования отсутствует. Если вы хотите, чтобы новый класс объединял свойства нескольких, породите классы-предки один от другого или включите в один класс несколько полей, соответствующих другим желаемым классам.

В приложениях VCL для платформы .NET существует еще один тип классов, для которых дальнейшее наследование запрещается. В объявлении таких классов присутствует ключевое слово `sealed`:

```
TSealedClass = class sealed(TClass)
end;
```

Полиморфизм

Рассмотрим внимательно следующий пример (листинг 3.4). Пусть у нас имеются некое обобщенное поле для хранения данных — класс `TField` и три его потомка для хранения строк, целых и вещественных чисел.

Листинг 3.4. Пример полиморфизма

```
type
 TField = class
  function GetData:string; virtual; abstract;
 end;
 TStringField = class(TField)
  FData : string;
  function GetData: string; override;
 end;
 TIntegerField = class(TField)
  FData : Integer;
  function GetData: string;override;
 end;
 TExtendedField = class(TField)
  FData : Extended;
  function GetData: string;override;
 end;
...
 function TStringField.GetData;
  begin
```

```
    Result := FData;
   end;
 function TIntegerField.GetData;
  begin
    Result := IntToStr(FData);
  end;
 function TExtendedField.GetData;
  begin
    Result:= FloatToStrF(FData, ffFixed, 7, 2);
  end;
....
procedure ShowData(AField : TField);
begin
  Form1.Label1.Caption := AField.GetData;
end;
```

В этом примере классы содержат разнотипные поля данных `FData` и только-то и "умеют", что сообщить о значении этих данных текстовой строкой (при помощи метода `GetData`). Внешняя по отношению к ним процедура `ShowData` получает объект в виде параметра и показывает эту строку.

Правила контроля *соответствия типов* (typecasting) языка Delphi гласят, что *объекту как указателю на экземпляр объектного типа может быть присвоен адрес любого экземпляра любого из дочерних типов*. В процедуре `ShowData` параметр описан как `TField` — это значит, что в нее можно передавать объекты классов и `TStringField`, и `TIntegerField`, и `TExtendedField`, и любого другого потомка класса `TField`.

Но какой (точнее, чей) метод `GetData` при этом будет вызван? Тот, который соответствует классу *фактически* переданного объекта. Этот принцип называется *полиморфизмом*, и он, пожалуй, представляет собой наиболее важный козырь ООП.

Допустим, вы имеете дело с некоторой совокупностью явлений или процессов. Чтобы смоделировать их средствами ООП, нужно выделить их самые общие, типовые черты. Те из них, которые не изменяют своего содержания, должны быть реализованы в виде статических методов. Те же, которые изменяются при переходе от общего к частному, лучше облечь в форму виртуальных методов.

Основные, "родовые" черты (методы) нужно описать в классе-предке и затем перекрывать их в классах-потомках. В нашем примере программисту, пишущему процедуру вроде `ShowData`, важно лишь, что любой объект, пере-

данный в нее, является потомком `Tfield`, и он умеет сообщить о значении своих данных (выполнив метод `GetData`). Если, к примеру, такую процедуру скомпилировать и поместить в динамическую библиотеку, то эту библиотеку можно будет раз и навсегда использовать без изменений, хотя будут появляться и новые, неизвестные в момент ее создания классы-потомки `TField`!

Наглядный пример полиморфизма дает среда Delphi. В ней имеется класс `TComponent`, на уровне которого сосредоточены определенные "правила" взаимодействия компонентов между собой и со средой разработки. Следуя этим правилам, можно порождать от `TComponent` свои компоненты, настраивая Delphi на решение специальных задач.

Теперь, надеемся, стало более или менее ясно, какие преимущества ООП позволили ему стать основным способом разработки серьезного программного обеспечения. Те, кто начинал программировать еще для Windows 3.0, наверняка помнят, сколько усилий требовалось при написании совершенно тривиального кода. Сейчас для того же самого в Delphi достаточно буквально пары щелчков мышью! На самом деле именно сложность программирования для Windows стала катализатором внедрения ООП.

Методы

Из предыдущего материала читатели узнали, что функционирование объектов обеспечивается разными типами методов, которые различаются особенностями реализации механизма наследования. Теперь настало время рассмотреть эти методы более подробно.

Абстрактными называются методы, которые определены в классе, но не содержат никаких действий, никогда не вызываются и обязательно должны быть переопределены в потомках класса. Абстрактными могут быть только виртуальные и динамические методы. В Delphi такие методы объявляются с помощью одноименной директивы. Она указывается при описании метода:

```
procedure NeverCallMe; virtual; abstract;
```

При этом никакого кода для этого метода писать не нужно. Вызов метода `NeverCallMe` приведет к созданию исключительной ситуации `EAbstractError` (исключительным ситуациям посвящена *глава 5*).

Пример с классом `TField` (см. листинг 3.4) поясняет, для чего нужны абстрактные методы. В данном случае класс `TField` используется не сам по себе, его основное предназначение — быть родоначальником иерархии конкретных классов-"полей" и дать возможность абстрагироваться от частностей. Хотя параметр процедуры `ShowData` и описан как `TField`, но если передать в нее объект этого класса, произойдет исключительная ситуация вызова абстрактного метода.

Статические методы, а также любые поля в объектах-потомках ведут себя одинаково: вы можете без ограничений перекрывать старые имена и при этом изменять тип методов. Код нового статического метода полностью перекрывает (заменяет собой) код старого метода (листинг 3.5).

Листинг 3.5. Пример перекрытия методов

```
type
T1stObj = class
  i : Extended;
  procedure SetData(AValue: Extended);
  end;
T2ndObj = class(T1stObj)
  i : Integer;
  procedure SetData(AValue: Integer);
 end;
....
procedure T1stObj.SetData;
begin
i := 1.0;
end;
procedure T2ndObj.SetData;
begin
i := 1;
inherited SetData(0.99);
end;
```

Здесь *разные* методы с именем `SetData` присваивают значения *разным* полям с именем `i`. Перекрытое (одноименное) поле предка недоступно в потомке; поэтому, конечно, два одноименных поля с именем `i` — это нонсенс; так сделано только для примера.

Примечание

В практике программирования принято присваивать всем идентификаторам в программе (в том числе полям объектов) осмысленные названия. Это существенно облегчит работу с исходным кодом не только другим разработчикам, но и вам.

В отличие от поля, внутри других методов перекрытый метод доступен при указании зарезервированного слова `inherited`. По умолчанию все методы

объектов являются статическими — их адрес определяется еще на стадии компиляции проекта, поэтому они вызываются быстрее всего. Может быть, читателю еще не ясно, для чего упоминается этот факт. Просто запомните его, он понадобится при сравнении статических и виртуальных методов.

Принципиально отличаются от статических *виртуальные* и *динамические* методы. Они должны быть объявлены путем добавления соответствующей директивы `virtual` или `dynamic`. С точки зрения наследования, методы этих двух видов одинаковы: они могут быть перекрыты в дочернем классе только одноименными методами, имеющими тот же тип.

Если задуматься над рассмотренным примером (листинг 3.4), становится ясно, что у компилятора нет возможности определить класс объекта, фактически переданного в процедуру `ShowData`. Нужен механизм, позволяющий определить это прямо во время выполнения. Такой механизм называется *поздним связыванием* (late binding).

Естественно, что этот механизм должен быть каким-то образом связан с передаваемым объектом. Для этого используются *таблица виртуальных методов* (Virtual Method Table, VMT) и *таблица динамических методов* (Dynamic Method Table, DMT).

Разница между виртуальными и динамическими методами заключается в особенности поиска адреса. Когда компилятор встречает обращение к виртуальному методу, он подставляет вместо прямого вызова по конкретному адресу код, который обращается к VMT и извлекает оттуда нужный адрес.

Такая таблица есть для каждого класса (объектного типа). В ней хранятся адреса *всех* виртуальных методов класса, независимо от того, унаследованы ли они от предка или перекрыты в данном классе. Отсюда и достоинства и недостатки виртуальных методов: они вызываются сравнительно быстро, однако для хранения указателей на них в таблице VMT требуется большой объем памяти.

Динамические методы вызываются медленнее, но позволяют более экономно расходовать память. Каждому динамическому методу системой присваивается уникальный индекс. В таблице динамических методов класса хранятся индексы и адреса *только тех* динамических методов, которые описаны в данном классе. При вызове динамического метода происходит поиск в этой таблице; в случае неудачи просматриваются таблицы DMT всех классов-предков в порядке иерархии и, наконец, класс `TObject`, где имеется стандартный обработчик вызова динамических методов. Экономия памяти налицо.

Для перекрытия и виртуальных, и динамических методов служит директива `override`, с помощью которой (и только с ней!) можно переопределять оба этих типа методов. Листинг 3.6 иллюстрирует это.

Листинг 3.6. Использование директивы `override`

```
type
  TFirstClass = class
  FMyField1: Integer;
  FMyField2: Longint;
  procedure StatMethod;
  procedure VirtMethod1; virtual;
  procedure VirtMethod2; virtual;
  procedure DynaMethod1; dynamic;
  procedure DynaMethod2; dynamic;
  end;
  TSecondClass = class(TMyObject)
  procedure StatMethod;
  procedure VirtMethod1; override;
  procedure DynaMethod1; override;
end;
var
  Obj1: TFirstClass;
  Obj2: TSecondClass;
```

Первый из методов в примере создается заново, остальные два — перекрываются. Попытка применить директиву `override` к статическому методу вызовет ошибку компиляции.

Примечание

Будьте внимательны: попытка перекрытия с директивой не `override`, а `virtual` или `dynamic` приведет на самом деле к созданию нового одноименного метода.

Перегрузка методов

Есть еще одна, совершенно особенная разновидность методов — *перегружаемые*.

Эту категорию методов нельзя назвать антагонистом двух предыдущих, поскольку и статические, и виртуальные, и динамические методы могут быть перегружаемыми. *Перегрузка методов нужна, чтобы выполнить одинаковые или похожие действия с разнотипными данными.*

Изменим пример (см. листинг 3.5), иллюстрировавший статические методы, рассмотрим следующий фрагмент кода (листинг 3.7).

Листинг 3.7. Пример перегрузки методов

```
type
T1stObj = class
  FExtData : Extended;
  procedure SetData(AValue: Extended);
  end;
T2ndObj = class(T1stObj)
  FIntData : Integer;
  procedure SetData(AValue: Integer);
 end;
...
var T1: T1stObj;
    T2 : T2ndObj;
```

В этом случае попытка вызова из объекта `T2` методов

```
...
T2.SetData(1.0);
T2.SetData(1);
...
```

вызовет ошибку компиляции на первой из двух строк. Для компилятора внутри `T2` статический метод с параметром типа `extended` перекрыт, и он его "не признает". Где же выход из сложившегося положения? Переименовать один из методов, например создать `SetIntegerData` и `SetExtendedData`? Можно, но если методов не два, а, скажем, сто, моментально возникнет путаница. Сделать методы виртуальными? Нельзя, так как тип и количество параметров в одноименных виртуальных методах должны в точности совпадать. Теперь для этого существуют перегружаемые методы, объявляемые при помощи директивы `overload`:

```
type
T1stObj = class
  FExtData : Extended;
  procedure SetData(AValue: Extended);overload;
  end;
T2ndObj = class(T1stObj)
```

```
  FIntData : Integer;
  procedure SetData(AValue: Integer); overload;
 end;
```

Объявив метод `SetData` перегружаемым, в программе можно использовать обе его реализации одновременно. Это возможно потому, что компилятор определяет тип передаваемого параметра (целый или с плавающей точкой) и в зависимости от этого подставит вызов соответствующего метода: для целочисленных данных — метод объекта `T2ndObj`, для данных с плавающей точкой — объекта `T1stObj`.

Можно перегрузить и виртуальный (динамический) метод. Надо только в этом случае добавить директиву `reintroduce`:

```
type
T1stObj = class
  FExtData : Extended;
  procedure SetData(AValue: Extended); overload; virtual;
  end;
T2ndObj = class(T1stObj)
  FIntData : Integer;
  procedure SetData(AValue: Integer); reintroduce; overload;
 end;
```

На перегрузку методов накладывается ограничение — нельзя перегружать методы, находящиеся в области видимости `published`, т. е. те, которые будут использоваться в Инспекторе объектов. Области видимости обсуждаются далее.

Области видимости

При описании нового класса важен разумный компромисс. С одной стороны, требуется скрыть от других методы и поля, представляющие собой внутреннее устройство класса (для этого и придуманы свойства). Маловажные детали на уровне пользователя объекта будут бесполезны и только помешают целостности восприятия.

С другой стороны, если слишком ограничить того, кто будет порождать классы-потомки, и не обеспечить ему достаточный набор инструментальных средств и свободу маневра, то он и не станет работать с таким классом.

В модели объектов языка Delphi существует механизм доступа к составным частям объекта, определяющий области, где ими можно пользоваться (*области видимости*). Поля и методы могут относиться к четырем группам (секциям), отличающимся областями видимости. Методы и свойства могут быть *общими* (секция `public`), *личными* (секция `private`), *защищенными*

(секция protected) и *опубликованными* (секция published). Есть еще и пятая группа, automated, она ранее служила для создания объектов COM.

Перечислим области видимости, определяемые первыми тремя директивами.

- Поля, свойства и методы секции public не имеют ограничений на видимость. Они доступны из других функций и методов объектов, как в данном модуле, так и во всех прочих, ссылающихся на него.
- Поля, свойства и методы, находящиеся в секции private, доступны только в методах класса и в функциях, содержащихся в том же модуле, что и описываемый класс. Такая директива позволяет полностью скрыть детали внутренней реализации класса. Свойства и методы из секции private можно изменять, и это не будет сказываться на программах, работающих с объектами этого класса. Единственный способ для кого-то другого обратиться к ним — переписать заново созданный вами модуль (если, конечно, доступны исходные тексты).
- Поля, свойства и методы секции protected также доступны только внутри модуля с описываемым классом. Но — и это главное — они доступны в классах, являющихся потомками данного класса, в том числе и в других модулях. Такие элементы особенно необходимы для разработчиков новых компонентов — потомков уже существующих. Оставляя свободу модернизации класса, они все же скрывают детали реализации от того, кто только пользуется объектами этого класса.

Рассмотрим пример (листинг 3.8), иллюстрирующий три варианта областей видимости.

Листинг 3.8. Пример задания областей видимости методов

```
unit First;
interface


type
TFirstObj = class
private
 procedure Method1;
protected
 procedure Method2;
public
 procedure Method3;
end;
```

```
unit Second;
interface
uses First;

type
TSecondObj = class(TFirstObj)
 procedure Method4;
end;
```

```
procedure TestProc1;

implementation

uses dialogs;
var AFirstObj: TFirstObj;

procedure TestProc1;
begin
 AFirstObj := TFirstObj.Create;
 AFirstObj.Method1; {допустимо}
 AFirstObj.Method2; {допустимо}
 AFirstObj.Method3; {допустимо}
 AFirstObj.Free;
end;

procedure TFirstObj.Method1;
begin
 ShowMessage('1');
end;
procedure TFirstObj.Method2;
begin
 ShowMessage('2');
 Method1;
end;
procedure TFirstObj.Method3;
begin
 ShowMessage('3');
 Method2;
end;

end.
```

```
procedure TestProc2;

implementation

var AFirstObj : TFirstObj;
   ASecondObj: TSecondObj;

procedure TSecondObj.Method4;
begin
  Method1; {недопустимо -
произойдет ошибка компиляции}
  Method2; {допустимо}
  Method3; {допустимо}
end;

procedure TestProc2;
begin
 AFirstObj := TFirstObj.Create;
 AFirstObj.Method1; {недопустимо}
 AFirstObj.Method2; {недопустимо}
 AFirstObj.Method3; {допустимо}
 AFirstObj.Free;

 ASecondObj := TSecondObj.Create;
 ASecondObj.Method1; {недопустимо}
 ASecondObj.Method2; {допустимо}
 ASecondObj.Method3; {допустимо}
 ASecondObj.Free;
end;

end.
```

Если к этому примеру добавить модуль `Third` и попробовать вызвать методы классов `TFirstObj` и `TSecondObj` оттуда, то к числу недоступных будет отнесен и `Method2` — он доступен только в том модуле, в котором описан.

Наконец, область видимости, определяемая четвертой директивой — `published`, имеет особое значение для интерфейса визуального проектиро-

вания Delphi. В этой секции должны быть собраны те свойства объекта, которые будут видны не только во время исполнения приложения, но и из среды разработки. Публиковать можно свойства большинства типов, за исключением старого типа `real` (теперь он называется `real48`), свойств типа "массив" и некоторых других. Все свойства компонентов, доступные через Инспектор объектов, являются их опубликованными свойствами. Во время выполнения такие свойства общедоступны, как и `public`.

Три области видимости — `private`, `protected`, `public` — как бы упорядочены по возрастанию видимости методов. В классах-потомках можно повысить видимость методов и свойств, но не понизить ее. При описании дочернего класса можно переносить методы и свойства из одной сферы видимости в другую, не переписывая их заново и даже не описывая — достаточно упомянуть о нем в другом месте (листинг 3.9).

Листинг 3.9. Пример описания дочернего класса

```
type
 TFirstObj = class
 private
  FNumber: Integer;
 protected
  property Number: Integer read: FNumber;
 end;
...
 TSecondObj = class(TFirstObj)
 published
  property Number;
 end;
```

Если какое-либо свойство объекта из состава VCL принадлежит к области `public`, вернуть его в `private` невозможно. Напротив, обратная процедура широко практикуется в Delphi. У многих компонентов (например, `TEdit`) есть предок (в данном случае `TCustomEdit`), который отличается только отсутствием опубликованных свойств. Так что, если вы хотите создать новый редактирующий компонент, порождайте его на базе `TCustomEdit` и публикуйте только те свойства, которые считаете нужными. Разумеется, если вы поместили свойство в область `private`, "достать" его оттуда в потомках возможности уже нет.

При разработке приложений VCL для платформы .NET существуют особенности использования областей видимости.

Начнем с того, что в .NET области видимости определены несколько иначе:

- поля, свойства и методы области видимости `public` не имеют ограничений на видимость;
- поля, свойства и методы области видимости `assembly` доступны только в сборке, в которой они объявлены;
- поля, свойства и методы области видимости `family` доступны в классе, в котором они объявлены, и в классах, являющихся потомками данного класса;
- поля, свойства и методы области видимости `private`, доступны только в методах класса и в функциях класса.

Также существуют две комбинации областей видимости:

- `assembly` и `family` обеспечивает видимость как в классе, так и в сборке;
- `assembly` или `family` обеспечивает видимость или в классе, или в сборке.

Для того чтобы классы приложений VCL для .NET соответствовали концепции видимости .NET, в Delphi дополнительно введены две секции:

- `strict private` — поля, свойства и методы доступны только в классе, в котором они объявлены;
- `strict protected` — поля, свойства и методы доступны только в классе, в котором они объявлены, и во всех дочерних классах.

Таким образом, можно составить таблицу соответствия областей видимости Delphi и .NET (табл. 3.1).

Таблица 3.1. *Соответствие областей видимости Delphi и .NET*

Delphi	.NET
`public`	`public`
`protected`	`assembly` или `family`
`private`	`assembly`

Объект изнутри

Теперь, когда мы разобрались с основными определениями и механизмами ООП, настало время более подробно изучить, что представляет собой объект и как он работает. Ясно, что каждый экземпляр класса содержит отдельную копию всех его полей. Очевидно, что где-то в его недрах есть указатели на таблицу виртуальных методов и таблицу динамических методов. А что еще

там имеется? И как происходит вызов методов? Вернемся к примеру из *разд. "Полиморфизм" данной главы* (см. листинг 3.4). Рассмотрим следующий фрагмент кода (листинг 3.10).

Листинг 3.10. Иллюстрация работы объекта

```
type
  TFirstClass = class
  FMyField1: Integer;
  FMyField2: Longint;
  procedure StatMethod;
  procedure VirtMethod1; virtual;
  procedure VirtMethod2; virtual;
  procedure DynaMethod1; dynamic;
  procedure DynaMethod2; dynamic;
  end;
  TSecondClass = class(TMyObject)
  procedure StatMethod;
  procedure VirtMethod1; override;
  procedure DynaMethod1; override;
end;
  Obj1: TFirstClass;
  Obj2: TSecondClass;
```

На рис. 3.1 показано, как будет выглядеть внутренняя структура рассмотренных в нем объектов.

Первое поле каждого экземпляра каждого объекта содержит указатель на его класс. Класс как структура состоит из двух частей. Начиная с адреса, на который ссылается указатель на класс, располагается таблица виртуальных методов. Напомним, что она содержит адреса всех виртуальных методов класса, включая унаследованные от предков. Длина таблиц VMT объектов `Obj1` и `Obj2` одинакова — по два элемента (8 байт). Перед таблицей виртуальных методов расположена специальная структура, содержащая дополнительную служебную информацию. В ней содержатся данные, полностью характеризующие класс: его имя, размер экземпляра, указатели на класс-предок, имя класса и т. д. На рис. 3.1 она показана одним блоком, а ее содержимое расшифровано ниже.

Одно из полей структуры содержит адрес таблицы динамических методов класса (DMT). Таблица имеет следующий формат: сначала слово, содержащее количество элементов таблицы, затем — слова, соответствующие индек-

сам методов. Нумерация индексов начинается с −1 и идет по убывающей. После индексов идут собственно адреса динамических методов. Обратите внимание, что DMT объекта `Obj1` состоит из двух элементов , `Obj2` — из одного, соответствующего перекрытому методу `DynaMethod1`. В случае вызова `Obj2.DynaMethod2` индекс не будет найден в таблице DMT `Obj2` и произойдет обращение к DMT `Obj1`. Именно так экономится память благодаря динамическим методам.

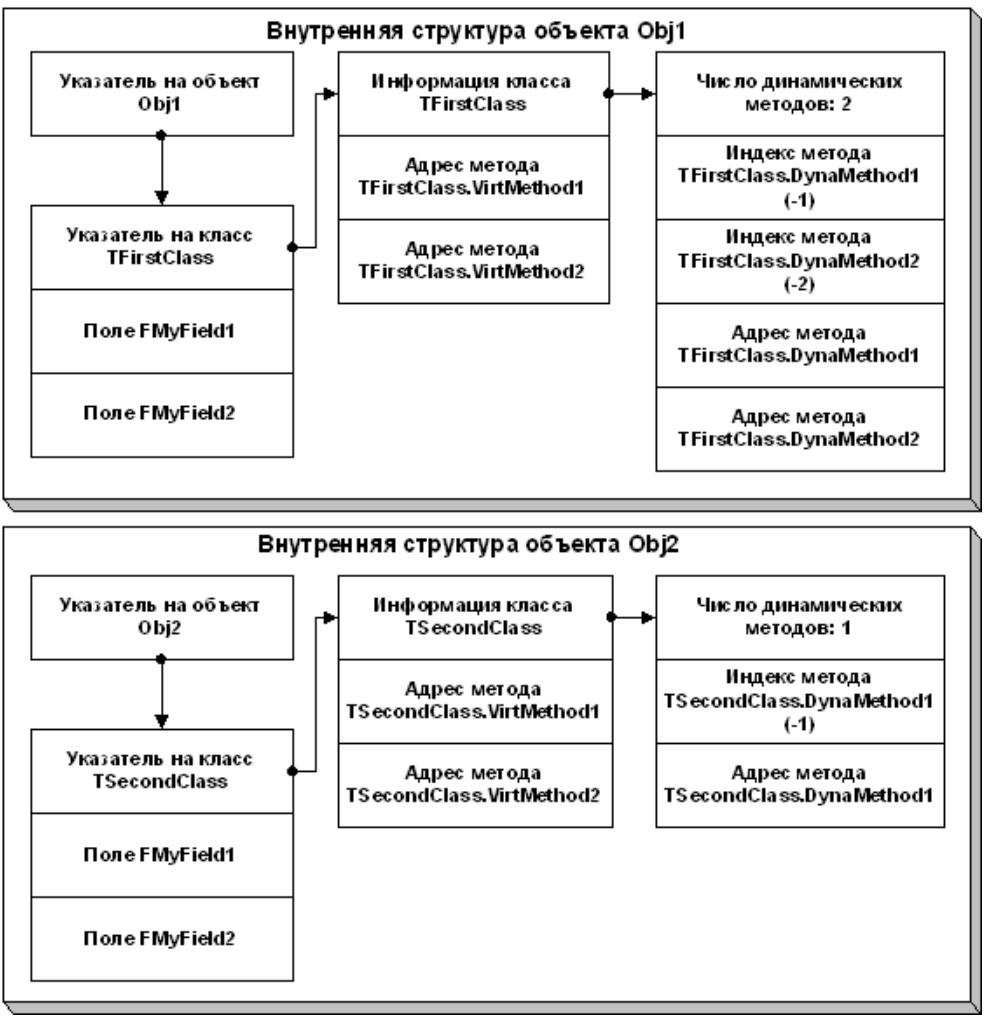


Рис. 3.1. Внутренняя структура объектов `Obj1` и `Obj2`

В языке Delphi определены два оператора `is` и `as`, неявно обращающиеся к таблице динамических методов. Оператор `is` предназначен для проверки совместимости по присваиванию экземпляра объекта с заданным классом. Выражение вида:

```
AnObject is TObjectType
```

принимает значение `True`, только если объект `AnObject` совместим по присваиванию с классом `TObjectType`, т. е. является объектом этого класса или одного из классов, порожденных от него. Кстати, определенная проверка происходит еще при компиляции: если формально объект и класс несовместимы, то компилятор выдаст ошибку в этом операторе.

Оператор `as` введен в язык специально для приведения объектных типов. С его помощью можно рассматривать экземпляр объекта как принадлежащий к другому совместимому типу:

```
with ASomeObject as TAnotherType do ...
```

От стандартного способа приведения типов с помощью конструкции `TAnotherType(ASomeObject)` использование оператора `as` отличается наличием проверки на совместимость типов во время выполнения (как в операторе `is`): попытка приведения к несовместимому типу приводит к возникновению исключительной ситуации `EInvalidCast` *(см. главу 5)*. После применения оператора `as` сам объект остается неизменным, но вызываются те его методы, которые соответствуют присваиваемому классу.

Очень полезным может быть оператор `as` в методах-обработчиках событий. Для обеспечения совместимости в 99% случаев источник события `Sender` имеет тип `TObject`, хотя, как правило, им является форма или другие компоненты. Поэтому, чтобы иметь возможность пользоваться их свойствами, применяют оператор `as`:

```
(Sender as TControl).Caption := 'Thanks!';
```

Вся информация, описывающая класс, создается и размещается в памяти на этапе компиляции. Возникает резонный вопрос: а нельзя ли получить доступ к ней, не создавая экземпляр объекта? Да, можно. Доступ к информации класса вне методов этого класса можно получить, описав соответствующий указатель, который называется *указателем на класс*, или *указателем на объектный тип* (class reference). Он описывается при помощи зарезервированных слов `class of`. Например, указатель на класс `TObject` описан в модуле SYSTEM.PAS и называется `TClass`:

```
type
  TObject = class;
  TClass = class of TObject;
```

Аналогичные указатели уже описаны и для других важных классов. Вы можете использовать в своей программе `TComponentClass`, `TControlClass` и т. п.

Указатели на классы тоже подчиняются правилам приведения объектных типов. Указатель на класс-предок может ссылаться и на любые дочерние классы, обратное невозможно (листинг 3.11).

Листинг 3.11. Пример использования указателей на класс

```
type
 TFirst = class
 ...
 end;
 TSecond = class(TFirst)
 ...
 end;
 TFirstClass = class of TFirst;
 TSecondClass = class of TSecond;
var
 AFirst : TFirstClass;
 ASecond : TSecondClass;
begin
 AFirst := TSecond; {допустимо}
 ASecond := TFirst; {недопустимо}
end.
```

С указателем на класс тесно связано понятие *методов класса*. Такие методы можно вызывать без создания экземпляра объекта — с указанием имени класса, в котором они описаны. Перед описанием метода класса нужно поставить зарезервированное слово `class` (листинг 3.12).

Листинг 3.12. Пример описания метода класса

```
type
  TMyObject = class(TObject)
    class function GetSize: string;
  end;
var
  MyObject: TMyObject;
  AString: string;
```

```
begin
  AString := TMyObject.GetSize;
  MyObject := TMyObject.Create;
  AString := MyObject.GetSize;
end.
```

Разумеется, методы класса не могут использовать значения, содержащиеся в полях класса: ведь экземпляра-то не существует! Возникает вопрос: для чего нужны такие методы?

Важнейшие методы класса определены в самом `TObject`, они как раз и позволяют, не углубляясь во внутреннюю структуру класса, извлечь оттуда практически всю необходимую информацию.

Резюме

В этой главе рассмотрены основы объектно-ориентированного программирования в Delphi. Объект обладает свойствами и методами, которые позволяют изменять значения свойств. Знание основ ООП необходимо для изучения всех глав не только этой части, но и всех последующих. Ведь компоненты Delphi — это объекты, размещенные в специальной библиотеке VCL. А ни одна глава этой книги не обходится без описания возможностей тех или иных компонентов.

Рассмотренные в этой главе возможности объектов применяются при создании исходного кода приложений Delphi.

Глава 4

Библиотека визуальных компонентов VCL и ее базовые классы

Все классы библиотеки визуальных компонентов произошли от группы базовых классов, которые лежат в основе иерархии VCL. Самый общий предок компонентов — это класс `TObject`, инкапсулирующий простейший объект. Как известно *(см. главу 3)*, каждый объект наследует свойства и методы родительского класса. К объекту можно добавить новые свойства и методы, но нельзя удалить унаследованные. Объект-наследник в свою очередь может стать родительским для нового класса, который унаследует возможности всех своих предков.

Поэтому иерархия базовых классов VCL продумана чрезвычайно тщательно — ведь на их основе создано все множество компонентов, используемых в Delphi. Особое место среди базовых классов, помимо `TObject`, занимают `TComponent` (от него происходят все компоненты) и `TControl` (от него происходят все элементы управления).

В этой главе рассматриваются иерархия базовых классов и их возможности. Представленные здесь сведения помогут разобраться с основными механизмами функционирования компонентов и послужат справочным материалом для тех, кто создает собственные объекты и элементы управления.

Иерархия базовых классов

В основе всего многообразия классов и компонентов, имеющихся в Delphi, лежат всего лишь пять базовых классов (рис. 4.1). Они обеспечивают выполнение основных функций любого объекта — будь это стандартный компонент VCL или специализированный объект, выполняющий некоторые операции в приложении.

Благодаря механизму наследования свойств и методов потомки базовых классов умеют "общаться" друг с другом, работают в среде разработки, взаи-

модействуя с Палитрой компонентов и Инспектором объектов, распознаются операционной системой как элементы управления и окна.

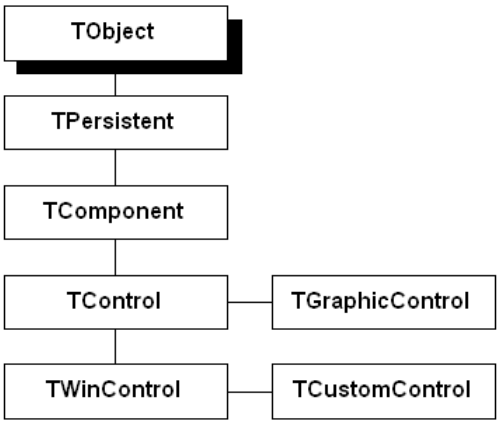


Рис. 4.1. Иерархия базовых классов VCL

В основе иерархии классов лежит класс `TObject`. Он обеспечивает выполнение важнейших функций "жизнедеятельности" любого объекта. Благодаря ему каждый класс получает в наследство механизмы создания экземпляра объекта и его уничтожения.

Обычно разработчик даже не задумывается о том, как объект будет создан и что необходимо сделать для его корректного уничтожения. Компоненты VCL создаются и освобождают занимаемые ресурсы автоматически. Иногда программисту приходится создавать и удалять объекты самостоятельно. Но даже в этом случае ему достаточно вызвать соответствующие конструктор и деструктор (листинг 4.1).

Листинг 4.1. Создание и удаление объекта

```
var SomeList: TStrings;
...
 SomeList := TStrings.Create;
...
 SomeList.Free;
```

За кажущейся простотой этих операций скрывается довольно сложная реализация указанных процессов. Практически весь исходный код класса `TObject` написан на ассемблере для обеспечения наибольшей эффективности операций, которые будут выполняться в каждом его потомке.

Кроме этого, класс `TObject` обеспечивает создание и хранение информации об экземпляре объекта и обслуживание очереди сообщений.

Класс `TPersistent` происходит непосредственно от класса `TObject`. Он обеспечивает своих потомков возможностью взаимодействовать с другими объектами и процессами на уровне данных. Его методы позволяют передавать данные в потоки, а также обеспечивают взаимодействие объекта с Инспектором объектов.

Класс `TComponent` является важнейшим для всех компонентов. Непосредственно от него можно создавать любые невизуальные компоненты. Механизмы, реализованные в классе `TComponent`, обеспечивают взаимодействие компонента со средой разработки, главным образом с Палитрой компонентов и Инспектором объектов. Благодаря возможностям этого класса компоненты начинают работать на форме проекта уже на этапе разработки.

Класс `TControl` происходит от класса `TComponent`. Его основное назначение — обеспечить функционирование визуальных компонентов. Каждый визуальный компонент, произошедший от `TControl`, наделяется основными признаками элемента управления. Благодаря этому каждый визуальный компонент умеет работать с GUI (Graphic User Interface — графический интерфейс пользователя ОС) и отображать себя на экране.

Класс `TWinControl` расширяет возможности разработчиков по созданию элементов управления. Он наследует от `TControl` и обеспечивает создание оконных элементов управления.

На основе класса `TWinControl` создан еще один дополнительный класс — `TCustomControl`. Он обеспечивает создаваемые на его основе компоненты возможностями по использованию канвы — специального объекта, предназначенного для отображения графики *(см. главу 12)*.

Класс `TCustomControl` является общим предком для целой группы классов, обеспечивающих создание различных нестандартных типов оконных (получающих фокус) элементов управления Windows (редакторов, списков и т. д.).

Для создания неоконных (не получающих фокус) элементов управления предусмотрен класс `TGraphicControl`, являющийся потомком класса `TControl`.

В целом иерархия базовых классов обеспечивает полноценную работу разработчиков в Delphi, позволяя проектировать любые типы приложений.

Далее мы остановимся на основных свойствах и методах базовых классов, выделяя только те, которые могут пригодиться в реальной работе. Часть из них доступна в Инспекторе объектов, часть может употребляться в программном коде.

Класс *TObject*

Класс `TObject` является родоначальником всей иерархии существующих в Delphi классов VCL. Он реализует функции, которые обязательно будет выполнять любой объект, который может быть создан в среде разработки. Учитывая большое разнообразие возможных областей применения объектов в процессе создания приложений, можно сказать, что круг общих для всех классов операций весьма невелик.

В первую очередь это создание экземпляра объекта и его уничтожение. Любой объект выполняет эти две операции в обязательном порядке.

Процесс создания объекта включает выделение области адресного пространства, установку указателя на экземпляр объекта, задание начальных значений свойств и выполнение установочных действий, связанных с назначением объекта. В общем случае две последние операции могут не выполняться.

Указатель на экземпляр объекта передается в переменную объектного типа, которая в дальнейшем будет идентифицировать объект в программном коде приложения. В приведенном ранее фрагменте кода (см. листинг 4.1) переменная объектного типа `SomeList` объявлена как экземпляр типа `TStrings`. При создании экземпляра этого типа конструктор `Create` возвращает в переменную `SomeList` указатель на выделенную для нового объекта область памяти. Для этого применяется метод `NewInstance`, который вызывается в конструкторе автоматически:

```
class function NewInstance: TObject; virtual;
```

Объект `TObject` обеспечивает выполнение этого процесса для любого порожденного от него объекта. А уже внутри конструктора, который унаследован от `TObject`, можно предусмотреть инициализацию переменных и выполнение дополнительных операций.

Объявление конструктора выглядит следующим образом:

```
constructor Create;
```

В конструкторах потомков это объявление может перекрываться, но при необходимости вызвать конструктор предка используется оператор `inherited`:

```
constructor TSomeObject.Create;
begin
 inherited Create;
...
end;
```

Для уничтожения экземпляра объекта в `TObject` предназначены методы `Destroy` и `Free`:

```
destructor Destroy; virtual;
procedure Free;
```

Как видно из объявления, настоящим деструктором является метод `Destroy`. Он обеспечивает освобождение всех занимаемых экземпляром объекта ресурсов. Обычно при создании новых классов деструктор всегда перекрывается для того, чтобы корректно завершить работу с данными.

Примечание

Обратите внимание, что обычно унаследованный конструктор вызывается в первую очередь. Это необходимо для того, чтобы выполнить все нужные операции при создании объекта до инициализации его собственных свойств. При уничтожении объекта обычно сначала выполняются завершающие операции и только в самом конце вызывается унаследованный деструктор, чтобы выполнить собственно уничтожение объекта.

Для уничтожения объектов вместо деструктора рекомендуется метод `Free`, который просто вызывает деструктор, но перед этим проверяет, чтобы указатель на экземпляр объекта был не пустым (не был равен `Nil`). Это позволяет избежать серьезных ошибок.

Если объект является владельцем других объектов (например, форма владеет всеми размещенными на ней компонентами), то его метод `Free` автоматически вызовет эти же методы для всех объектов. Поэтому при закрытии формы разработчик избавлен от необходимости заботиться об уничтожении всех компонентов.

Для освобождения занимаемой объектом памяти деструктор автоматически вызывает метод `FreeInstance`:

```
procedure FreeInstance; virtual;
```

Каждый объект должен содержать некоторую информацию о себе, которая используется приложением и средой разработки. Поэтому класс `TObject` содержит ряд методов, обеспечивающих представление этой информации в потомках.

Метод

```
class function ClassInfo: Pointer;
```

возвращает указатель на таблицу информации времени выполнения (RTTI).

Функция

```
class function ClassName: ShortString;
```

возвращает имя типа объекта, которое может служить для его идентификации. Например, один метод-обработчик щелчка на кнопке может работать с несколькими типами компонентов кнопок:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
 if Sender is TBitBtn
```

```
  then TBitBtn(Sender).Enabled := False;
 if Sender is TSpeedButton
  then TSpeedButton(Sender).Down := True;
end;
```

Метод

```
class function ClassNameIs(const Name: string): Boolean;
```

позволяет определить, является ли данный объект того типа, имя которого передано в параметре `Name`. В случае положительного ответа функция возвращает `True`.

Как известно, программирование для Windows основано на событиях. Каждое приложение и каждый программный объект должны уметь реагировать на сообщение о событиях и, в свою очередь, рассылать сообщения. В выполнении этих операций заключается третья общая для всех объектов функция.

Метод

```
procedure Dispatch(var Message); virtual;
```

осуществляет обработку сообщений, поступающих объекту. Он определяет, сможет ли объект обработать сообщение при помощи собственных обработчиков событий. В случае отсутствия таких методов сообщение передается аналогичному методу `Dispatch` класса-предка (если он есть).

Класс `TObject` имеет предопределенный обработчик событий:

```
procedure DefaultHandler(var Message); virtual;
```

Кроме рассмотренных здесь методов класс `TObject` имеет еще несколько методов, которые необходимы в основном для организации взаимодействия объекта со средой разработки.

В целом, класс `TObject` удобен для создания на его основе некоторых простых классов для использования в приложениях.

Класс *TPersistent*

"Persistent" в переводе с английского означает "устойчивый", "постоянный". Что же такого постоянного в одноименном классе? Ответ таков: виртуальный метод

```
procedure Assign(Source: TPersistent);
```

Этот важнейший метод осуществляет копирование содержимого одного объекта (`Source`) в другой (`Self`, т. е. в объект, вызвавший метод `Assign`). При этом объект-получатель остается самим собой, чего нельзя достигнуть простым присваиванием переменных объектного типа:

```
FirstObject := SecondObject;
```

Ведь в этом случае указатель на одну область адресного пространства, содержащую экземпляр класса (объект), замещается указателем на другую область адресного пространства, содержащую другой объект.

Метод `Assign` позволяет продублировать объект — присвоить одному объекту значения всех свойств другого. При этом объекты не обязательно должны быть одного и того же класса; более того, они не обязательно должны находиться в отношениях "родитель-потомок"! Данный метод тем и хорош, что обеспечивает полиморфное присвоение. Конструкция

```
Clipboard.Assign( Picture );
```

позволяет скопировать содержимое картинки `Picture` в папку обмена Windows (объект `Clipboard`). Какова здесь логика? Известно, что в папку обмена можно поместить растровую картинку, текст, метафайл, мультимедийные данные и т. п. Метод `Assign` класса `TClipboard` переписан таким образом, чтобы реализовать присвоение (т. е. реальное перемещение в папку обмена) всех этих данных.

```
procedure TClipboard.Assign(Source: TPersistent);
begin
 if Source is TPicture
  then AssignPicture(TPicture(Source))
  else
   if Source is TGraphic
    then AssignGraphic(TGraphic(Source))
    else inherited Assign(Source);
end;
```

Для обеспечения взаимодействия потомков класса `TPersistent` со средой разработки предназначен метод

```
function GetNamePath: string; dynamic;
```

Он возвращает имя объекта для передачи его в Инспектор объектов.

Для взаимодействия с потоками при загрузке и сохранении компонентов есть следующий метод:

```
procedure DefineProperties(Filer: TFiler); virtual;
```

Класс `TPersistent` никогда не используется напрямую, от него порождаются потомки, которые должны уметь передавать другим объектам значения своих свойств, но не являться при этом компонентами.

Класс *TComponent*

Класс `TComponent` является предком всех компонентов VCL. Он является основой для создания невизуальных компонентов и реализует основные ме-

ханизмы, которые обеспечивают функционирование любого компонента. В нем появляются первые свойства, которые отображаются в Инспекторе объектов. Это свойство

```
property Name: TComponentName;
```

Оно содержит имя экземпляра компонента, которое служит для идентификации компонента в приложении.

Примечание

Тип `TComponentName` представляет собой обычную строку:

```
type TComponentName = type string;
```

Свойство

```
property Tag: Longint;
```

является вспомогательным и не влияет на работу компонента. Оно отдано на откуп разработчику, который может присваивать ему значения по своему усмотрению. Например, это свойство удобно для дополнительной идентификации компонентов.

Для компонентов существует своя иерархия, поэтому в классе введен механизм учета и управления компонентами, для которых данный компонент является владельцем. Свойства и методы, которые отвечают за управление, приведены в табл. 4.1.

Таблица 4.1. *Методы и свойства для управления списком компонентов*

Свойство (метод)	Описание
`property Components[Index: Integer]: TComponent;`	Содержит индексированный список указателей всех компонентов, для которых данный компонент является владельцем (`Owner`)
`property ComponentCount: Integer;`	Число подчиненных компонентов
`property Owner: TComponent;`	Указывается, какой компонент является владельцем данного
`property ComponentIndex: Integer;`	Индекс данного компонента в списке владельца
`procedure InsertComponent (AComponent: TComponent);`	Вставляет компонент `AComponent` в список
`procedure RemoveComponent (AComponent: TComponent);`	Удаляет компонент `AComponent` из списка
`procedure FindComponent(AName: string): TComponent;`	Осуществляет поиск компонента по имени `AName`

Таблица 4.1 *(окончание)*

Свойство (метод)	Описание
`procedure DestroyComponents;`	Предназначен для уничтожения всех компонентов, подчиненных данному

Очень важное свойство

```
type TComponentState = set of (csLoading, csReading, csWriting,
csDestroying, csDesigning, csAncestor, csUpdating, csFixups,
csFreeNotification, csInline, csDesignInstance);
property ComponentState: TComponentState;
```

дает представление о текущем состоянии компонента. В табл. 4.2 описаны возможные значения свойства. Состояние может измениться в результате получения компонентом некоторого сообщения, действий разработчика, выполнения акции и т. д. Это свойство активно используется средой разработки.

Таблица 4.2. *Возможные состояния компонента*

Состояние	Описание
`csLoading`	Устанавливается при загрузке компонента из потока
`csReading`	Устанавливается при чтении значений свойств из потока
`csWriting`	Устанавливается при записи значений свойств в поток
`csDestroying`	Устанавливается при уничтожении компонента
`csDesigning`	Состояние разработки. Устанавливается во время работы с формой при разработке
`csAncestor`	Устанавливается при переносе компонента на форму. Для перехода в это состояние должно быть уже установлено состояние `csDesigning`
`csUpdating`	Устанавливается при изменении значений свойств и отображения результата на форме-владельце. Для перехода в это состояние должно быть уже установлено состояние `csAncestor`
`csFixups`	Устанавливается, если компонент связан с компонентом другой формы, которая еще не загружена в среду разработки
`csFreeNotification`	Если этот флаг поднимается, другие компоненты, связанные с данным, уведомляются о его уничтожении

Таблица 4.2 *(окончание)*

Состояние	Описание
csInline	Определяет компонент верхнего уровня в иерархии. Используется для обозначения корневого объекта в разворачивающихся свойствах
csDesignInstance	Определяет корневой компонент на этапе разработки

Для обеспечения работы механизма действий (Actions, *см. главу 9*) предназначен метод

```
function ExecuteAction(Action: TBasicAction): Boolean; dynamic;
```

Он вызывается автоматически при необходимости выполнить акцию, предназначенную для данного компонента.

На уровне класса `TComponent` обеспечена поддержка COM-интерфейсов `IUnknown` и `IDispatch`.

Через свойство

```
property ComObject: IUnknown;
```

вы можете обеспечить использование методов этих интерфейсов.

Таким образом, класс `TComponent` имеет все для использования в качестве предка для создания собственных невизуальных компонентов.

Базовые классы элементов управления

Вслед за `TComponent` в иерархии базовых классов (см. рис. 4.1) располагается группа из трех классов, которые обеспечивают создание различных визуальных компонентов. Визуальные компоненты — это разнообразные стандартные для Windows и специальные (созданные разработчиками Borland) элементы управления.

Понятно, что визуальные компоненты должны уметь отобразить себя на экране монитора и реагировать на целый ряд новых событий (реакция на мышь и клавиатуру, движение курсора и т. д.). Для этого в них встроен специальный механизм, обеспечивающий взаимодействие компонентов с графической подсистемой ОС (GUI).

Существует несколько типов элементов управления, которые существенно отличаются по своим возможностям и поведению. Каждому типу соответствует собственный класс иерархии.

Класс `TWinControl` обеспечивает в Delphi работу с оконными элементами управления. Главное отличие оконного элемента управления от любых дру-

гих — наличие дескриптора окна `hWnd`. Дескриптор окна — это специальный идентификатор, который операционная система присваивает всем объектам, которые должны обладать свойствами окна. Если элемент управления имеет дескриптор окна, то он должен уметь выполнять следующие операции:

- получать и передавать фокус управления во время выполнения приложения;
- воспринимать управляющие воздействия от мыши и клавиатуры;
- уметь размещать на себе другие элементы управления.

Оконными элементами управления являются не только формы, но и практически все стандартные элементы управления Windows: и списки, и редакторы имеют дескриптор окна.

Итак, все визуальные компоненты происходят от класса `TWinControl`. Однако нестандартные элементы управления имеют еще одного общего предка. Это класс `TCustomControl`. Он существенно облегчает использование элементов управления, так как позволяет управлять отрисовкой компонента путем использования специального класса `TCanvas` — так называемой канвы *(см. главу 12)* вместо обращения к системным функциям GUI.

Для обеспечения создания обычных (не оконных) элементов управления непосредственно от класса `TControl` порожден класс `TGraphicControl`. Его потомки не могут получать фокус, но используют для визуализации канву.

Класс *TControl*

Класс `TControl` является базовым для всех визуальных компонентов и инкапсулирует механизмы отображения компонента на экране. В нем используется множество новых свойств и методов. Недаром в Delphi в Инспекторе объектов появилась категоризация методов и свойств (рис. 4.2). Большинство из них вводятся как раз в классах `TControl` и `TWinControl`.

Рассмотрим только важнейшие свойства и методы по категориям.

Местоположение и размер элемента управления

Для определения местоположения и размеров визуального компонента введены два опубликованных свойства для задания координат левого верхнего угла:

```
property Top: Integer;
property Left: Integer;
```

и два опубликованных свойства для определения размеров:

```
property Height: Integer;
property Width: Integer;
```

Рис. 4.2. Категории свойств визуального компонента. Для представлении их в таком виде нужно отметить флажок **By Category** в пункте меню **Arrange** всплывающего меню Инспектора объектов

Значения свойств задаются в пикселах. Местоположение определяется в системе координат рабочей области владельца данного компонента. Начало отсчета находится в левом верхнем углу. Оси направлены вправо и вниз. Под рабочей областью понимается та часть площади владельца (формы, панели), которая предназначена для размещения дочерних элементов. Эти свойства доступно как на этапе разработки, так и во время выполнения.

Свойство

```
property ClientOrigin: TPoint;
```

содержит координаты левого верхнего угла элемента управления в системе координат экрана. Координаты любой точки можно пересчитать в экранные при помощи метода

```
function ClientToScreen(const Point: TPoint): TPoint;
```

и наоборот

```
function ScreenToClient(const Point: TPoint): TPoint;
```

Для приведения компонента в соответствие текущим значениям указанных свойств преднизначен метод

```
procedure AdjustSize; dynamic;
```

Параметры рабочей области компонента определяются следующими свойствами.

```
property ClientHeight: Integer;
```

определяет высоту рабочей области в пикселах.

```
property ClientWidth: Integer;
```

определяет ширину рабочей области в пикселах. Кому-то будет удобнее пользоваться свойством

```
property ClientRect: TRect;
```

значение которого есть ни что иное, как (`0,0,.ClientWidth, ClientHeight`).

Если разработчику неизвестны текущие параметры рабочей области, то он может воспользоваться следующими методами.

Функция

```
function GetClientOrigin: TPoint; virtual;
```

возвращает координаты левого верхнего угла рабочей области.

Функция

```
function GetClientRect: TRect; virtual;
```

возвращает размеры прямоугольника рабочей области.

Выравнивание элемента управления

Для выравнивания компонента в рабочей области его владельца (обычно это форма) применяется свойство

```
property Align: TAlign;
```

Тип `TAlign` объявлен следующим образом:

```
type TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient);
```

При значении `alNone` выравнивание отсутствует. При следующих четырех значениях выравнивание осуществляется по соответствующей стороне. Значение `alClient` приводит к тому, что элемент управления изменяет свои размеры до размеров рабочей области владельца.

Свойство

```
property Anchors: TAnchors;
type TAnchors = set of TAnchorKind;
type TAnchorKind = (akTop, akLeft, akRight, akBottom);
```

обеспечивает фиксацию элемента управления по сторонам владельца. "Якорь" можно установить по одной, двум, трем или четырем сторонам. При задании якоря по любой стороне расстояние между стороной и элементом

управления сохраняется неизменным. Комбинируя якоря для сторон, можно добиться различных вариантов поведения компонента при изменении размеров владельца.

Если по вертикали или горизонтали якорь не установлен вообще, то при изменении размеров владельца компонент остается на том же месте с учетом изменившегося размера.

Если по вертикали или горизонтали установлены оба якоря, то при изменении размеров владельца размер элемента управления изменяется таким образом, чтобы расстояния до сторон владельца остались неизменными.

Свойство

```
property AutoSize: Boolean;
```

обеспечивает изменение размеров компонента в соответствии с размерами его содержимого (текста, изображения, списка, иерархического дерева и т. д.).

Внешний вид элемента управления

Для определения цвета компонента предусмотрено свойство

```
property Color: TColor;
```

При нахождении указателя мыши над компонентом его изображение может изменяться в соответствии со значением свойства

```
property Cursor: TCursor;
```

Для текста компонента шрифт обычно задается свойством

```
property Font: TFont;
```

Сложный класс `TFont`, задающий все характеристики шрифта, подробно рассматривается в *главе 19*.

Свойство

```
property DesktopFont: Boolean;
```

определяет возможность отображения текста шрифтом, используемый ОС для представления текста в значках.

Сам текст задается свойством

```
type TCaption = string;
property Text: TCaption;
```

Длину текста можно определить при помощи функции

```
function GetTextLen: Integer;
```

Она возвращает число символов в тексте.

Элемент управления можно сделать видимым или невидимым. Для этого применяется свойство

```
property Visible: Boolean;
```

Этих же результатов можно достичь методами `Show` (компонент видим) и `Hide` (компонент невидим).

Опубликованное свойство

```
property Hint: string;
```

содержит текст ярлыка — однострочной подсказки, которая появляется в маленькой рамке при остановке мыши на элементе управления.

Для управления ярлыком есть свойство

```
property ShowHint: Boolean;
```

При значении `True` ярлык начинает работать, при значении `False` — выключается.

Для каждого элемента управления можно создать собственное всплывающее меню. Ссылка на экземпляр класса `TPopupMenu` *(см. главу 9)*, инкапсулирующего такое меню, хранится в свойстве

```
property PopupMenu: TPopupMenu;
```

Текущее состояние элемента управления определяется свойством `ControlState`:

```
type TControlState = set of (csLButtonDown, csClicked, csPalette,
csReadingState, csAlignmentNeeded, csFocusing, csCreating, csPaintCopy,
csCustomPaint, csDestroyingHandle, csDocking,);
property ControlState: TControlState;
```

Возможные значения свойства описаны в табл. 4.3.

Таблица 4.3. *Возможные состояния элемента управления*

Состояние	Описание
`csLButtonDown`	Левая кнопка мыши нажата, но еще не отпущена. Используется для реализации события `OnMouseDown`
`csClicked`	Левая кнопка мыши нажата, но еще не отпущена. Используется для реализации события `OnClick`
`csPalette`	Состояние соответствует режиму изменения палитры. Это реакция на сообщение WM_PALETTCHANGED
`csReadingState`	Осуществляется чтение значений свойств из потока
`csAlignmentNeeded`	Осуществляется выравнивание компонента
`csFocusing`	Элемент управления получает фокус

Таблица 4.3 *(окончание)*

Состояние	Описание
csCreating	Элемент управления и его дочерние элементы создаются
csPaintCopy	Отрисовывается копия элемента управления
csCustomPaint	Элемент управления выполняет нестандартные операции отрисовки, заданные разработчиком
CsDestroyingHandle	Указатель на объект элемента управления уничтожается
CsDocking	Элемент управления находится в режиме докования

В зависимости от совокупности установленных свойств элемент управления может соответствовать одному из возможных стилей, который задается свойством

```
ype TControlStyle = set of (csAcceptsControls, csCaptureMouse,
csDesignInteractive, csClickEvents, csFramed, csSetCaption, csOpaque,
csDoubleClicks, csFixedWidth, csFixedHeight, csNoDesignVisible,
csReplicatable, csNoStdEvents, csDisplayDragImage, csReflector,
csActionClient, csMenuEvents);

property ControlStyle: TControlStyle;
```

Доступность элемента управления в целом определяется свойством

```
property Enabled: Boolean;
```

При значении `True` элемент управления полностью работоспособен. При значении `False` элемент управления неактивен и отображается серым цветом.

Для получения контекста устройства `HDC` элемента управления используется метод

```
function GetDeviceContext(var WindowHandle: HWnd): HDC; virtual;
```

Набор свойств и методов класса `TWinControl` обеспечивает функционирование механизма перетаскивания (Drag and Drop) и механизма докования (Drag and Dock).

Связь с родительским элементом управления

Механизм связывания визуального компонента с родительским компонентом (владельцем) позволяет автоматически задавать для нового элемента управления некоторые свойства, отвечающие за его внешний вид. В результате все дочерние элементы управления для одного родительского (формы, панели) будут выглядеть одинаково оформленными.

Родительский компонент задается свойством

```
property Parent: TWinControl;
```

Для каждого дочернего элемента управления можно задать значения нескольких свойств:

```
property ParentBiDiMode: Boolean;
property ParentColor: Boolean;
property ParentFont: Boolean;
property ParentShowHint: Boolean;
```

Каждое из них управляет одной характеристикой визуализации элемента.

Метод

```
function HasParent: Boolean; override;
```

необходим для того, чтобы определить, имеется ли у компонента владелец вообще.

В классе `TControl` впервые появляются методы-обработчики событий, которые обеспечивают передачу в элемент действия мыши, докование и перетаскивание.

Класс *TWinControl*

Класс `TWinControl` обеспечивает реализацию оконных элементов управления. Напомним, что оконный элемент управления имеет системный дескриптор окна `hWnd`. Однако оконными элементами являются не только формы и диалоги, но и большинство стандартных элементов управления.

Новые механизмы, инкапсулированные в классе, обеспечивают выполнение характерных для оконных элементов функций: прием и передачу фокуса, отклик на действия мышью и ввод с клавиатуры и т. д. Рассмотрим основные свойства и методы класса.

Дескриптор окна содержится в свойстве

```
property Handle: HWND;
```

При создании оконного элемента управления вызывается метод

```
procedure CreateParams(var Params: TCreateParams); virtual;
```

который заполняет структуру `TCreateParams` необходимыми значениями:

```
type
  TCreateParams = record
    Caption: PChar;
    Style: DWORD;
    ExStyle: DWORD;
    X, Y: Integer;
    Width, Height: Integer;
```

```
    WndParent: HWND;
    Param: Pointer
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;
```

Создание дескриптора окна для элемента управления осуществляется методом

```
procedure CreateHandle; virtual;
```

Операционная система создает дескриптор окна только вместе с самим окном. Поэтому метод `CreateHandle` только создает окно, а для присваивания свойству `Handle` значения дескриптора окна вызывает метод `CreateWnd`.

Для передачи фокуса между элементами управления на одной форме часто используется клавиша <Tab>. Порядок перемещения фокуса между элементами определяется свойством

```
type TTabOrder = -1..32767;
property TabOrder: TTabOrder;
```

В первую очередь фокус передается компоненту с минимальным значением свойства. Далее — по возрастанию значения. При переносе компонента на форму это значение задается автоматически в соответствии с числом компонентов на форме.

Компонент можно заставить не откликаться на клавишу <Tab>. Для этого свойству

```
property TabStop: Boolean;
```

необходимо присвоить значение `False`.

Для передачи фокуса прямо элементу управления применяется метод

```
procedure SetFocus; virtual;
```

Чтобы узнать, имеет ли элемент управления фокус в настоящее время, используется метод

```
function Focused: Boolean; dynamic;
```

Все оконные элементы имеют рамку по контуру (впрочем, она может быть не видна). Ее параметры задаются группой свойств.

```
property BevelEdges: TBevelEdges;
```

задает, какие стороны входят в рамку.

```
property BevelInner: TBevelCut;
property BevelOuter: TBevelCut;
```

задают внешний вид рамки.

```
property BevelKind: TBevelKind;
```

определяет стиль рамки.

```
property BevelWidth: TBevelWidth;
```

задает размер рамки.

Свойство

```
property Brush: TBrush;
```

определяет параметры кисти (цвет и заполнение), которой рисуется фон элемента.

Оконный элемент может содержать другие компоненты. Для управления ими применяется индексированный список указателей, представляющих свойство

```
property Controls[Index: Integer]: TControl;
```

Общее число дочерних элементов управления содержится в свойстве

```
property ControlCount: Integer;
```

Внешний вид оконного элемента определяется свойством

```
property Ctl3D: Boolean
```

При значении `True` элемент управления имеет трехмерный вид. Иначе элемент выглядит плоским.

Для вызова темы контекстной помощи для конкретного элемента управления предназначено свойство

```
type THelpContext = -MaxLongInt..MaxLongInt;
property HelpContext: THelpContext;
```

Значение свойства должно соответствовать номеру темы в файле помощи.

В классе `TWinControl` добавлена возможность выбора редакторов способа ввода (Input Method Editor, IME), которые позволяют приспособить стандартную раскладку клавиатуры для символьных языков для ввода нестандартных символов (иероглифов и т. п.). Редакторы IME представляют собой специально устанавливаемое в операционной системе ПО. Имя такого редактора задается в свойстве `ImeName`. Режим работы редактора определяется свойством `ImeMode`.

В классе `TWinControl` добавлено еще несколько методов-обработчиков событий, обеспечивающих реакцию на ввод с клавиатуры, получение и потерю фокуса.

Класс *TCustomControl*

Класс `TCustomControl` предназначен для создания на его основе нестандартных оконных элементов управления. Процесс визуализации в нем упрощен

за счет использования специального класса TCanvas, инкапсулирующего канву *(см. главу 12)*.

Доступ к канве осуществляется через свойство

```
property Canvas: TCanvas;
```

Отрисовка элемента управления осуществляется методом

```
procedure PaintWindow(DC: HDC); override;
```

после получения сообщения WM_PAINT.

Возможности этого класса унаследовали классы TPanel, TGroupBox, TStringGrid и т. д.

Класс *TGraphicControl*

Класс TGraphicControl предназначен для создания на его основе визуальных компонентов, не получающих фокус в процессе выполнения приложения. Так как непосредственным предком класса является класс TControl, то потомки TGraphicControl умеют реагировать на управляющие воздействия мышью.

Наглядный пример элемента управления, которому не нужно получать фокус, — это компонент TLabel, предназначенный для отображения текста или TImage, предназначенный для визуализации изображений.

Для визуализации элементов управления на основе этого класса необходима канва, инкапсулированная в классе TCanvas.

Доступ к канве осуществляется через свойство

```
property Canvas: TCanvas;
```

Отрисовка элемента управления осуществляется методом

```
procedure PaintWindow(DC: HDC); override;
```

после получения сообщения WM_PAINT.

Резюме

В этой главе рассмотрены важнейшие свойства и методы базовых классов, лежащих в основе VCL. Познакомившись с возможностями этих классов, вы тем самым будете знать назначение многих свойств и методов в компонентах VCL. Поэтому материал этой главы будет полезен при изучении любых вопросов, рассматриваемых в этой книге в дальнейшем.

Глава 5

Обработка исключительных ситуаций

Каждому, кто писал более или менее сложные программы, интуитивно ясно, что такое обработка исключительных ситуаций (ИС). Любое взаимодействие с операционной системой на предмет получения ресурсов — места на диске, в памяти, открытие файла — может завершиться неудачно. Всякое вычисление может закончиться делением на ноль или переполнением. Дополнительный фактор возникновения исключительных ситуаций содержится в данных, к которым могут обращаться программы. Особенно это актуально в приложениях баз данных.

Платой за надежную работу программы в таких условиях служит введение многочисленных проверок, способных предотвратить некорректные действия в случае возникновения нештатной ситуации. Хорошо, если в конце очередной конструкции `if...then` можно просто поставить оператор `Exit`. Обычно же для корректного выхода из ситуации нужно отменить целую последовательность действий, предшествующих неудачному. Все это сильно запутывает программу, маскируя четкую структуру главного алгоритма.

При разработке приложений в Delphi программист имеет возможность использовать несколько механизмов, обеспечивающих обработку исключительных ситуаций. Это и специальные операторы языка Delphi, и классы, предназначенные для программирования реакции на ошибки.

Поэтому эта глава посвящена... нет, не тому, как писать безошибочно, а тому, как защищать приложения от воздействия неизбежно возникающих ошибок.

Исключительная ситуация как класс

Что же такое исключительная ситуация? Интуитивно понятно, что это — некое нештатное событие, могущее повлиять на дальнейшее выполнение программы. Компилятор Delphi генерирует код, который перехватывает любое такое нештатное событие, сохраняет необходимые данные о состоянии

программы и выдает разработчику... Что можно выдать в объектно-ориентированном языке программирования? Конечно же, объект. *С точки зрения языка Delphi исключительная ситуация — это объект.*

Вы можете получить и обработать этот объект, добавив в программу специальную языковую конструкцию (`try..except`). Если такая конструкция не предусмотрена, все равно исключение будет обработано — в недрах VCL есть соответствующие обработчики, окружающие все потенциально опасные места.

Чем же различаются между собой исключительные ситуации? Как отличить одну исключительную ситуацию от другой? Поскольку это объекты, они отличаются классом (объектным типом). В модуле SYSUTILS.PAS описан объектный тип `Exception`. Он является предком для всех других объектов — исключительных ситуаций (листинг 5.1).

Листинг 5.1. Описание класса `Exception`

```
Exception = class(TObject)

  private

    FMessage: string;

    FHelpContext: Integer;

  public

    constructor Create(const Msg: string);

    constructor CreateFmt(const Msg: string; const Args: array of const);

    constructor CreateRes(Ident: Integer); overload;

    constructor CreateRes(ResStringRec: PResStringRec); overload;

    constructor CreateResFmt(Ident: Integer; const Args: array of const);
overload;

    constructor CreateResFmt(ResStringRec: PResStringRec; const Args:
array of const); overload;

    constructor CreateHelp(const Msg: string; AHelpContext: Integer);

    constructor CreateFmtHelp(const Msg: string; const Args: array of
const;

      AHelpContext: Integer);

    constructor CreateResHelp(Ident: Integer; AHelpContext: Integer);
overload;

    constructor CreateResHelp(ResStringRec: PResStringRec; AHelpContext:
Integer); overload;
```

```
    constructor CreateResFmtHelp(ResStringRec: PResStringRec; const Args:
array of const; AHelpContext: Integer); overload;

    constructor CreateResFmtHelp(Ident: Integer; const Args: array of
const; AHelpContext: Integer); overload;

    property HelpContext: Integer read FHelpContext write FHelpContext;

    property Message: string read FMessage write FMessage;

  end;


  ExceptClass = class of Exception;
```

Как видно из приведенного описания класса `Exception`, у него имеется двенадцать (!) конструкторов, позволяющих задействовать при создании объекта текстовые строки из ресурсов приложения (имя включает строку `Res`), форматирование текста (включает `Fmt`), связь с контекстом справочной системы (включает `Help`).

Конструкторы, в названии которых встречается подстрока `Fmt`, могут вставлять в формируемый текст сообщения об ошибке значения параметров, как это делает стандартная функция `Format`:

```
If MemSize > Limit then
 raise EOutOfMemory.CreateFmt('Cannot allocate more than %d
bytes',[Limit]);
```

Если в названии присутствует подстрока `Res`, это означает, что текст сообщения будет загружаться из ресурсов приложения. Это особенно полезно при создании локализованных версий программных продуктов, когда нужно сменить язык всех сообщений, ничего не компилируя заново.

И, наконец, если в названии фигурирует подстрока `Help`, то такой конструктор инициализирует свойство `HelpContext` создаваемого объекта. Естественно, система помощи должна быть создана, и в ней должна иметься статья, связанная с этим контекстом. Теперь пользователь может затребовать помощь для данной ситуации, скажем, нажав клавишу <F1> в момент показа сообщения об ИС.

Тип `Exception` порождает многочисленные дочерние типы, соответствующие часто встречающимся случаям ошибок ввода/вывода, распределения памяти и т. п. Дерево исключительных ситуаций Delphi приведено на рис. 5.1.

Заметим, что `Exception` и его потомки представляют собой исключение из правила, предписывающего все объектные типы именовать с буквы `T`. Потомки `Exception` начинаются с `E`, например, `EZeroDivide`.

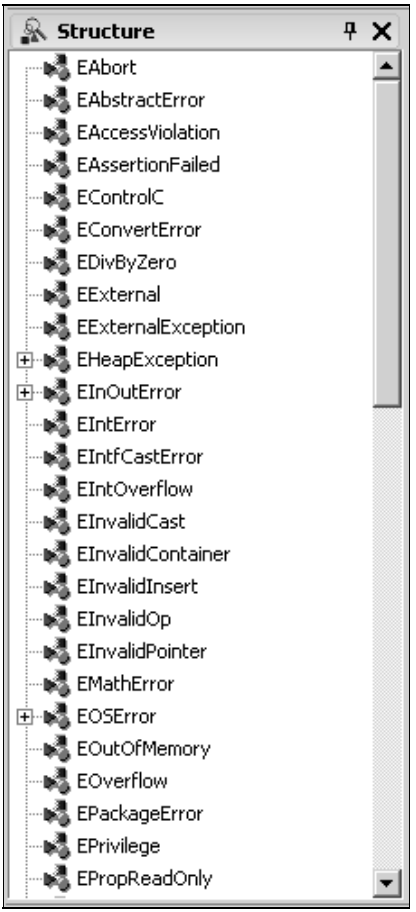


Рис. 5.1. Дерево объектов-исключительных ситуаций

Для экономии места потомки нескольких важных объектов не показаны. В приведенных табл. 5.1—5.3 содержится описание этих групп исключительных ситуаций.

Таблица 5.1. *ИС работы с памятью (порождены от* `EHeapException`*)*

Тип	Условие возникновения
`EOutOfMemory`	Недостаточно места в куче (памяти)
`EOutOfResources`	Нехватка системных ресурсов
`EInvalidPointer`	Недопустимый указатель (обычно `Nil`)

Таблица 5.2. *ИС целочисленной математики (порождены от* `EIntError`*)*

Тип	Условие возникновения
`EdivByZero`	Попытка деления на ноль (целое число)
`ErangeError`	Число или выражение выходит за допустимый диапазон
`EintOverflow`	Целочисленное переполнение

Таблица 5.3. *ИС математики с плавающей точкой (порождены от* `EMathError`*)*

Тип	Условие возникновения
`EInvalidOp`	Неверная операция
`EZeroDivide`	Попытка деления на ноль
`EOverflow`	Переполнение с плавающей точкой
`EUnderflow`	Исчезновение порядка
`EInvalidArgument`	Неверный аргумент математических функций

Вы можете самостоятельно инициировать исключительную ситуацию при выполнении тех или иных действий. Но, хотя синтаксис конструктора объекта `Exception` похож на конструкторы всех других объектов, создается он по-особенному.

Для этого используется оператор `raise`, за которым в качестве параметра должен идти экземпляр объекта типа `Exception`. Обычно сразу за оператором следует конструктор класса ИС:

```
raise EMathError.Create('');
```

создание и возбуждение исключительной ситуации можно разделить:

```
var E: EMathError;
begin
  E := EMathError.Create('');
  raise E;
end;
```

Оператор `raise` передает созданную исключительную ситуацию ближайшему блоку `try ..except`.

```
if C = 0 then
  raise EDivByZero.Create('Деление на ноль')
else
  A := B/C;
```

Самостоятельная инициализация ИС может пригодиться при программировании реакции приложения на ввод данных, для контроля значений переменных и т. д. В таких случаях желательно создавать собственные классы ИС, специально приспособленные для ваших нужд. Специально спроектированные исключительные ситуации полезны также при создании собственных объектов и компонентов. Так, многие важнейшие классы VCL — списки, потоки, графические объекты — сигнализируют о своих (или ваших?) проблемах созданием соответствующей ИС — `EListError`, `EInvalidGraphic`, `EPrinter` и т. д.

Самый важный отличительный признак объекта `Exception` — это все же класс, к которому он принадлежит. Именно факт принадлежности возникшей ИС к тому или иному классу говорит о том, что случилось. Если же нужно детализировать проблему, можно присвоить значение свойству `Message`. Если и этого мало, можно добавить в объект новые поля. Так, в ИС `EInOutError` (ошибка ввода/вывода) есть поле `ErrorCode`, значение которого соответствует произошедшей ошибке — запрету записи, отсутствию или повреждению файла и т. д. (листинг 5.2).

Листинг 5.2. Пример обработки исключительных ситуаций

```
try
.FileOpen( 'c:\myfile.txt', fmOpenWrite );
except
 on E: EInOutError do
 case E.ErrorCode of
  ERROR_FILE_NOT_FOUND {=2}: ShowMessage('Файл не найден !');
  ERROR_ACCESS_DENIED {=5}:  ShowMessage('Доступ запрещен!');
  ERROR_DISK_FULL {=112}:    ShowMessage('Диск переполнен!');
 end;
end;
```

Впрочем, ИС `EInOutError` возникают только тогда, когда установлена опция компилятора `{$IOCHECKS ON}` (или иначе `{$I+}`). В противном случае проверку переменной `IOResult` нужно делать самому.

Еще более "продвинутый" пример — ИС `EDBEngineError`. Она имеет свойства `ErrorCount` и свойство-массив `Errors`: одна операция с базой данных может породить сразу несколько ошибок.

Защитные конструкции языка Delphi

Для работы с объектами исключительных ситуаций существуют специальные конструкции языка Delphi: блоки `try..except` и `try..finally`. Они контролируют выполнение операторов, помещенных внутри блока до ключевого слова `except` или `finally`. В случае возникновения исключительной ситуации штатное выполнение вашей программы немедленно прекращается и управление передается операторам, идущим за указанными ключевыми словами. Если в вашей процедуре эти блоки отсутствуют, управление все равно будет передано ближайшему блоку, внутри которого возникла ситуация. А уж внутри VCL их предостаточно.

Хотя синтаксис двух видов блоков похож, но они *принципиально* отличаются назначением и решаемыми задачами. Поэтому вопрос, какой из них выбрать, не возникает: чтение нижеследующего материала, надеемся, убедит вас в пользе обоих.

Блок *try..except*

Для реакции на конкретный тип ситуации применяется блок `try..except`. Синтаксис его иллюстрирует листинг 5.3.

Листинг 5.3. Синтаксис `try..except`

```
try
 <Оператор>
 <Оператор>
 ...
 except
  on EException1 do  < Оператор обработки ИС типа EException1 >;
  on EException2 do  < Оператор >;
  ...
  else { }
  <Оператор> {обработчик прочих ИС}
end;
```

Выполнение блока начинается с секции `try`. При отсутствии исключительных ситуаций только она и выполняется. Секция `except` получает управление в случае возникновения ИС. После обработки происходит выход из защищенного блока, и управление обратно в секцию `try` не передается; выполняются операторы, стоящие после `end`.

Если вы хотите обработать любую ИС одинаково, независимо от ее класса, вы можете писать код прямо между операторами `except` и `end`. Но если обработка отличается, здесь можно применять набор директив `on...do`, которые задают реакцию приложения на определенную ситуацию. Каждая директива связывает ситуацию (`on...`), заданную своим именем класса, с группой операторов (`do...`). Листинг 5.4. содержит фрагмент кода, иллюстрирующий это.

Листинг 5.4. Применение директив `on...do`

```
U := 220.0;
R := 0;
try
  I := U / R;
except
  on EZeroDivide do MessageBox('Короткое замыкание!');
end;
```

В этом примере замена `if...then` на `try...except`, может быть, и не дала очевидной экономии кода. Однако если при решении, допустим, вычислительной задачи проверять на возможное деление на ноль приходится не один, а множество раз, то выигрыш от нового подхода неоспорим — достаточно одного блока `try..except` на все вычисления.

При возникновении ИС директивы просматриваются последовательно, в порядке их описания. Каждый тип исключительной ситуации, описанный после ключевого слова `on`, обрабатывается именно этим блоком: только то, что предусмотрено в нем, и будет являться реакцией на данную ситуацию.

Если при этом обработчик родительского класса стоит перед дочерним, последний никогда не получит управления (листинг 5.5).

Листинг 5.5. Неверное применение конструкции `on..do`

```
try
 i:=1;j:=0;
 k:=i div j;
...
except
 on EIntError do ShowMessage('IntError');
 on EDivByZero do ShowMessage('DivByZero');
 end;
```

В этом примере, хотя в действительности будет иметь место деление на ноль (`EDivByZero`), вы увидите сообщение, соответствующее родительскому классу `EIntError`. Но стоит поменять две конструкции `on..do` местами, и все придет в норму.

Если возникла ситуация, не определенная ни в одной из директив, выполняются те операторы, которые стоят после `else`. Если и их нет, то ИС считается необработанной и будет передана на следующий уровень обработки. Этим следующим уровнем может быть другой оператор `try..except`, который содержит в себе данный блок.

Примечание

Мы детально обсудили, что и как помещают в блок `try..except`. Но в ряде случаев можно... не размещать там ничего. Пустой блок применяют, когда хотят просто проигнорировать возникновение некоторых ситуаций. Скажем, у вас в программе предусмотрена некая обработка данных, которая может завершиться неудачно, но это не повлияет на дальнейшую работу, и пользователь может об этом не знать. В этой ситуации вполне уместно изолировать ее в пустом блоке `try..except`. Важно только не поместить туда больше кода, чем нужно — иначе с водой можно выплеснуть и ребенка.

Если вы не предусмотрели блоков обработки ИС в своем коде, это не должно привести к аварийному завершению всего приложения. Все места в VCL, где управление передается коду разработчика (в том числе, конечно, все обработчики событий всех компонентов), заключены в такие блоки. Но, увы, в Borland не знают о конкретных проблемах вашей программы, и максимум, что они могут сделать для вас — это проинформировать о типе и месте возникновения ИС. Стандартная обработка подразумевает вывод на экран панели текстового сообщения (из свойства `Exception.Message`) с указанием типа ошибки. Можно получить и развернутую информацию с именем модуля и адреса, где она имела место (рис. 5.2).

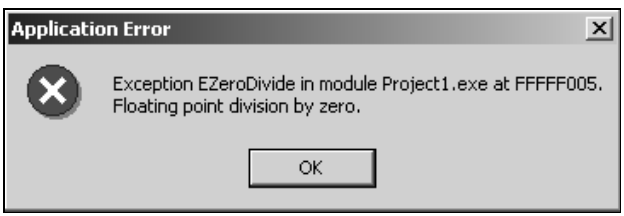


Рис. 5.2. Типовое окно сообщения об ошибке

Для этого нужно вызвать процедуру

```
procedure ShowException(ExceptObject: TObject; ExceptAddr: Pointer);
```

имеющуюся в модуле SYSUTILS.PAS. Подробнее о стандартной обработке рассказано далее в этой главе.

Если предусмотренной вами обработки ИС недостаточно, то можно продолжить ее дальше программно, при помощи оператора `raise`.

Этот оператор уже встречался нам при описании создания пользовательских ИС. Там за ним следовал вызов конструктора ИС. Здесь же конструктор опущен: возбуждаться будет уже существующий объект-ИС, приведший нас в блок:

```
...
sl:= TStringList.Create;
 try
   sl.LoadFromFile(AFileName);
except
   sl.Free;
   raise;
 end;
...
```

В этом примере в случае возникновения исключительной ситуации созданный список строк должен быть уничтожен. Сама же обработка предоставляется "вышестоящим инстанциям".

Блок *try..finally*

Параллельно с блоком `try..except` в языке существует и `try..finally`. Он соответствует случаю, когда необходимо возвратить выделенные программе ресурсы даже в случае аварийной ситуации. Синтаксис блока `try...finally` иллюстрирует листинг 5.6.

Листинг 5.6. Синтаксис блока `try...finally`

```
try
  <Оператор>
  <Оператор>
...
 finally
  <Оператор>
  ...
end;
```

Смысл этой конструкции можно описать одним предложением: операторы, стоящие после `finally`, выполняются *всегда*.

Следующие за `try` операторы выполняются в обычном порядке. Если за это время не возникло никаких ИС, далее выполняются те операторы, которые стоят после `finally`. В случае, если между `try` и `finally` произошла ИС, управление немедленно передается на оператор(ы) после `finally`, который(е) называется *кодом очистки*. Допустим, вы поместили после `try` операторы, которые должны выделить вам ресурсы системы (дескрипторы блоков памяти, файлов, контекстов устройств и т. п.). Тогда операторы, освобождающие их, следует поместить после `finally`, и ресурсы будут освобождены в любом случае. Блок `try..finally`, как можно догадаться, еще называется *блоком защиты ресурсов*.

Важно обратить внимание на такой факт: данная конструкция ничего не делает с самим объектом — исключительной ситуацией. Задача `try..finally` — только прореагировать на факт нештатного поведения программы и проделать определенные действия. Сама же ИС продолжает "путешествие" и вопрос ее обработки остается в повестке дня.

Блоки защиты ресурсов и обработчики ИС, как и другие блоки, могут быть вложенными. В этом простейшем примере каждый вид ресурсов системы защищается в отдельном блоке (листинг 5.7).

Листинг 5.7. Простейший пример обработки ИС

```
try
 Allocate1stResource;
 try
  Allocate2ndResource;
  SolveProblem;
 finally
  Free2ndResource;
 end;
finally
 Free1stResource;
end;
```

Можно также вкладывать обработчики друг в друга, предусмотрев в каждом специфическую реакцию на ту или иную ошибку (листинг 5.8).

Листинг 5.8. Обработка ИС вложенными блоками

```
var i,j,k : Integer;
begin
i := Round(Random);
j := 1 - i;
try
  k := 1 div i;
  try
    k := 1 div j;
  except
    On EDivByZero do
     ShowMessage('Вариант 1: j=0');
  end;
except
  On EDivByZero do
     ShowMessage('Вариант 2: i=0');
end;
end;
```

Но все же идеально правильный случай — это сочетание блоков двух типов. В один из них помещается общее (освобождение ресурсов в `finally`), в другой — особенное (конкретная реакция внутри `except`).

Использование исключительных ситуаций

Если произошла ошибка и возбуждена исключительная ситуация, то она будет обрабатываться по такому алгоритму.

1. Если ситуация возникла внутри блока `try..except`, то там она и будет обработана. Если ИС "продвинута" дальше при помощи оператора `raise`, а также, если она возникла в блоке `try..finally`, обработка продолжается.
2. Если программистом определен обработчик события `Application.OnException`, то он получит управление. Обработчик объявлен следующим образом:

   ```
   TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;
   ```

3. Если программист никак не определил реакцию на ИС, то будет вызван стандартный метод `ShowException`, который сообщит о классе и месте возникновения исключительной ситуации.

Пункты 2 и 3 реализуются в методе `TApplication.HandleException`. Собственно, выглядят они следующим образом:

```
if not (ExceptObject is EAbort) then
 if Assigned(FOnException) then
  FOnException(Sender, Exception(ExceptObject))
 else
  ShowException(Exception(ExceptObject));
```

Обработчик `OnException` нужен, если требуется выполнять одно и то же действие в любой исключительной ситуации, возникшей в вашем приложении. К примеру, назвать себя, указать координаты для обращения или предупредить, что это еще бета-версия (листинг 5.9).

Листинг 5.9. Пример обработки ИС с выдачей дополнительного сообщения

```
program Project1;
uses
  Forms,
  SysUtils, //добавлено вручную — там описан класс Exception
  Dialogs,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
 type
  TExceptClass = class
  public
  procedure GlobalExceptionHandler(Sender: TObject; E:Exception);
  end;
  procedure TExceptClass.GlobalExceptionHandler(Sender: TObject;
  E:Exception);
  begin
   ShowMessage('Произошла исключительная ситуация ' + E.ClassName
   + ': ' + E.Message
   + #13#10'Свяжитесь с разработчиками по тел. 222-33-44');
  end;
begin
  with TExceptClass.Create do
  begin
   Application.OnException := GlobalExceptionHandler;
   Application.Initialize;
```

```
   Application.CreateForm( TForm1, Form1 );
   Application.Run;
   Free;
  end;
end.
```

Здесь класс `TExceptClass` создается только для того, чтобы быть носителем метода `GlobalException`. Обработчик любого события — метод, и он должен относиться к какому-либо объекту. Поскольку он здесь нужен еще до инициализации форм приложения и других его составных частей, то и объект класса `TExceptClass` создается первым. Теперь пользователь знает, что благодарить за неожиданности нужно по указанному в сообщении об ошибке телефону разработчиков.

Примечание

Есть и более простой способ присвоить обработчик событию `Application.OnException`. Для этого поместите на форму компонент типа `TApplicationEvents` (категория Палитры инструментов **Additional**), роль которого — в предоставлении "визуального" доступа к свойствам невизуального объекта `TApplication`. Среди его событий есть и `OnException`.

Но как "пощупать" переданный при исключительной ситуации объект? Обычная конструкция

```
on EExceptionType do...
```

указывает на класс объекта, но не на конкретный экземпляр. Если во время обработки требуется доступ к свойствам этого экземпляра, его нужно поименовать внутри `on..do`, указав перед именем класса некий идентификатор:

```
on EZD: EZeroDivide do EZD.Message := 'Деление на ноль!';
```

Здесь возникшее исключение выступает под именем `EZD`. Можно изменить его свойства и отправить дальше:

```
 var  APtr : Pointer;
     Form1 : TForm;
 try
  APtr := Form1;
  with TObject(APtr) as TBitmap do;
 except
   on EZD: EInvalidCast do EZD.Message := . EZD.Message + 'ха-ха!';
      Raise;{ теперь обработка будет сделана в другом месте }
 end;
```

Но как поименовать исключительную ситуацию, не попавшую ни в одну из директив `on..do`? Или, может быть, в вашем обработчике вообще нет `on..do`, а поработать с объектом надо? Описанный выше путь здесь не подходит. Для этих случаев есть пара системных функций `ExceptObject` и `ExceptAddr`. К сожалению, эти функции инициализируются только внутри конструкции `try..except`; в `try..finally` работать с объектом — исключительной ситуацией не представляется возможным.

Протоколирование исключительных ситуаций

Часто нужно иметь подробный материал для анализа причин возникновения ИС. Разумно было бы записывать все данные о них в файл, чтобы потом прогнозировать ситуацию. Такой подход важен для программ, которые так или иначе будут отчуждены от разработчика: в случае возникновения непредвиденной ситуации это позволит ответить на вопросы "кто виноват?" и "что делать?" В следующем примере (листинг 5.10) предложен вариант реализации протоколирования ИС.

Листинг 5.10. Пример протоколирования ИС

```
const LogName : string = 'c:\appexc.log';
procedure LogException;
var fs: TFileStream; m : word;buf : array[0..511] of char;
begin
 if FileExists(LogName) then m := fmOpenReadWrite else m := fmCreate;
 fs := TFileStream.Create(LogName,m);
 fs.Seek(0,soFromEnd);
 StrPCopy(Buf,DateTimeToStr(Now)+'. ');
 ExceptionErrorMessage
(ExceptObject,ExceptAddr,@buf[StrLen(buf)],SizeOf(Buf)-StrLen(buf));
 StrCat(Buf,#13#10);
 fs.WriteBuffer(Buf, StrLen(buf) );
 fs.Free;
end;
procedure TForm1.Button1Click(Sender: TObject);
var x,y,z: real;
begin
try
 try
 x:=1.0;y:=0.0;
 z := x/y;
```

```
 except
  LogException;
  raise;
 end;
except
 on E:EIntError do ShowMessage('IntError');
 on E:EMathError do ShowMessage('MathError');
end;
end;
```

Здесь задачу записи информации об ИС решает процедура `LogException`. Она открывает файловый поток и пишет туда информацию, отформатированную при помощи уже упоминавшейся функции `ExceptionErrorMessage`. В качестве ее параметров выступают значения функций `ExceptObject` и `ExceptAddr`. К сформированной строке добавляется время возникновения ИС. Для каждого защищаемого блока кода создаются две вложенные конструкции `try..except`. Первая, внутренняя — для вас; в ней ИС протоколируется и продвигается дальше. Внешняя — для пользователя; именно в ней проводится анализ типа ИС и готовится сообщение.

В Delphi существует и расширенный вариант употребления оператора `raise`:

```
raise <экземпляр объекта типа Exception> [at <адрес>]
```

Естественно, объектный тип должен быть порожден от `Exception`. То, что в таком типе ничего не переопределено, не столь важно — главное, что в обработчике ИС можно отследить именно этот тип.

```
ELoginError = class(Exception);

 If LoginAttemptsNo > MaxAttempts then raise ELoginError.Create('Ошибка
 регистрации пользователя');
```

Конструкция `at <адрес>` необходима для того, чтобы изменить адрес, к которому привязывается возникшая ИС, в пределах одного блока обработки ИС.

Коды ошибок в исключительных ситуациях

Если ваше приложение уже готовится к продаже, если вы планируете его техническую поддержку, то пора задуматься о присвоении числовых кодов ошибкам, возникающим в нем. Сообщение типа "`Exception EZeroDivide in module MyNiceProgram at addr $0781BAB0`" годится для разработчика, пользователя же оно повергнет в полный ступор. Если он позвонит в вашу службу технической поддержки, то скорее всего, не сможет ничего объяс-

нить. Гораздо грамотнее дать ему уже "разжеванную" информацию, и в том числе — числовой код.

Один из путей решения этой проблемы — размещение сообщений об ошибках в ресурсах программы. Если же вы еще делаете и несколько национальных версий программы на разных языках, то этот путь — единственный.

"Классический" способ поместить текст в файл ресурсов — трехэтапный.

1. Создается исходный файл ресурсов с расширением rc, в который помещаются нужные строки с нужными номерами.
2. Файл компилируется компилятором ресурсов brcc32.exe (находится в папке bin в структуре папок Delphi). На выходе образуется одноименный файл с расширением res.
3. Файл включается в программу указанием директивы `$R`, например, `{$R mystrings.res}`.

Чтобы совместно использовать константы — номера ошибок в файле ресурсов и в коде на Delphi, вынесем их в отдельный включаемый файл с расширением inc:

```
const
 IOError = 1000;
 FileOpenError =  IOError + 1;
 FileSaveError =  IOError + 2;
 InternetError = 2000;
 NoConnectionError = InternetError + 1;
 ConnectionAbortedError = InternetError + 2;
```

Взглянув на файл, вы увидите, что ошибки в нем сгруппированы по категориям. Советуем вам поступить так же, разделив константы категорий промежутком в 1000 или даже 10 000.

Сам файл ресурсов может выглядеть так:

```
#include "strids.inc"
STRINGTABLE
 {
 FileOpenError, "File Open Error"
 FileSaveError, "File Save Error"
 NoConnectionError, "No Connection"
 ConnectionAbortedError, "Connection Aborted"
 }
```

"Вытащить" строку из ресурсов можно несколькими способами, но самый простой из них — просто по числовому идентификатору, переданному в функцию `LoadStr` (модуль `SysUtils`).

Код

```
ShowMessage( LoadStr(NoConnectionError) );
```

покажет сообщение "No Connection".

Если же строка используется при возбуждении ИС, то место идентификатору — в перекрываемом конструкторе `Exception.CreateRes`, один из вариантов которого работает подобно `LoadStr`:

```
if FileOpen('c:\myfile.txt", fmOpenRead) = INVALID_HANDLE_VALUE then
 raise EMyException.CreateRes( FileOpenError );
```

Таким образом, решена половина проблемы: возможным исключительным ситуациям присвоены номера, им в соответствие поставлен текст. Теперь о второй половине — как употребить этот номер в обработчике ИС.

Ясно, что нужно объявить свой класс ИС, включающий в себя свойство — код ошибки.

```
EExceptionWithCode = class( Exception )
 private
  FErrCode : Integer;
 public
  constructor CreateResCode( ResStringRec: PResStringRec );
  property ErrCode: Integer read FErrCode write FErrCode;
 end;
```

Тогда любой обработчик сможет к нему обратиться:

```
if E is EExceptionWithCode then
  ShowMessage( 'Error code: '+ IntToStr( EExceptionWithCode(E).ErrCode )
+ #13#10
  + 'Error text: ' + E.Message );
```

Присвоить свойству `ErrCode` значение можно двумя способами.

- Добавить к классу ИС еще один конструктор, содержащий код в качестве дополнительного параметра:

```
constructor EExceptionWithCode.CreateResCode( Ident: Integer );
begin
 FErrCode := Ident;
 inherited CreateRes( Ident );
end;
```

- Присвоить значение свойства в промежутке между созданием объекта ИС и его возбуждением:

```
var E: EExceptionWithCode;
begin
```

```
    E := EExceptionWithCode.CreateRes( NoConnectionError );
    E.ErrCode := NoConnectionError;
    Raise E;
   end;
```

Вот, казалось бы, последний штрих. Но как быть тем, кто заранее не заготовил файл ресурсов, а работает со строками, описанными в PAS-файлах? Если вы применяете оператор `resourcestring`, то помочь вам можно.

Начнем с рассмотрения ключевого слова `resourcestring`. Вслед за ним описываются текстовые константы. Но, в отличие от ключевого слова `const`, эти константы размещаются не в сегменте данных программы, а в ресурсах, и подгружаются оттуда по мере необходимости. Каждая такая константа воспринимается и обрабатывается как обычная строка. Но за каждой из них на самом деле стоит такая структура:

```
PResStringRec = ^TResStringRec;
  TResStringRec = packed record
    Module: ^Cardinal;
    Identifier: Integer;
  end;
```

Если вы еще раз посмотрите на список конструкторов объекта `Exception`, вы увидите, что те из них, которые работают с ресурсами, имеют перегружаемую версию с параметром типа `pResStringRec`. Вы угадали правильно: они — для строк из `resourcestring`. А взглянув на приведенную выше структуру, вы увидите в ней поле `Identifier`. Это именно то, что нам надо.

Чтобы у программиста, пользующегося `resourcestring`, голова не болела об уникальных идентификаторах ресурсных строк, среда Delphi берет на себя заботу об этом. Номера назначаются компилятором, начиная от 65535 (`SmallInt(-1)`) и ниже (если рассматривать номер как тип `SmallInt`, то выше): 65534, 65533 и т. п. Сначала в этом списке идут несколько сотен `resourcestring`-констант, описанных в VCL (из модулей, чье имя заканчивается на `const` или `consts`: `SysConst`, `DBConsts` и т. п.). Затем очередь доходит до пользовательских констант.

С одной стороны, отсутствие лишних забот — это большой плюс; с другой стороны, разработчик не может задать строкам те номера, какие хочет.

Все остальное почти ничем не отличается от работы с "самодельными" ресурсами. Так выглядит перегружаемая версия конструктора нашего объекта `EExceptionWithCode`:

```
constructor EExceptionWithCode.CreateResCode( ResStringRec: PResStringRec
);
begin
 FErrCode := ResStringRec^.Identifier;
```

```
 inherited CreateRes( ResStringRec );
end;
```

А так — возбуждение самой ИС:

```
resourcestring
 sError1 = 'Error 1';
..
Raise EExceptionWithCode.CreateResCode( PResStringRec(@sError1) );
```

Рис. 5.3. Результат обработки ИС типа `EExceptionWithCode`

Результат обработки показан на рис. 5.3.

Исключительная ситуация *EAbort*

Если вы внимательно просмотрели код системной процедуры `HandleException`, то увидели там упоминание класса `EAbort`. ИС `EAbort` служит единственным — и очень важным — исключением из правил обработки. Она называется "тихой" (Silent) и отличается тем, что для нее обработка по умолчанию не предусматривает вывода сообщений на экран. Естественно, все сказанное касается и порожденных от нее дочерних объектных классов.

Применение `EAbort` оправдано во многих случаях. Вот один из примеров. Пусть разрабатываются некоторая прикладная программа или некоторое семейство объектов, не связанное с VCL. Если в них возникает ИС, то нужно как-то известить об этом пользователя. А между тем прямой вызов для этого функции `ShowMessage` или даже `MessageBox` не всегда оправдан. Для маленькой и компактной динамической библиотеки не нужно тащить за собой громаду VCL. С другой стороны, в большом и разнородном проекте нельзя давать каждому объекту или подпрограмме самой общаться с пользователем. Если их разрабатывают разные люди, такой проект может превратиться в вавилонскую башню. Тут и поможет `EAbort`. Эта исключительная ситуация не создается системой — ее должен создавать и обслуживать программист.

Применение `EAbort` — реальная альтернатива многочисленным конструкциям `if..then` и тем более (упаси боже!) `goto`. Эта ИС не должна подменять

собой другие, вроде ошибки выделения памяти или чтения из файла. Она нужна, если вы сами видите, что сложились определенные условия и пора менять логику работы программы.

```
If LogicalCondition then Raise EAbort.Create('Condition 1');
```

Если не нужно определять сообщение, можно создать `EAbort` и проще — вызвав процедуру `Abort` (без параметров), содержащуюся в модуле SYSUTILS.PAS.

Функция *Assert*

Эта процедура и сопутствующая ей ИС `EAssertionFailed` специально перенесены в Delphi из языка C для удобства отладки. Синтаксис ее прост:

```
procedure Assert(expr : Boolean [; const msg: string]);
```

При вызове функции проверяется, чему равно значение переданного в нее булевого выражения `expr`. Если оно равно `True`, то ровным счетом ничего не происходит. Если же оно равно `False`, создается ИС `EAssertionFailed`. Все это было бы довольно тривиально с точки зрения уже изученного, если бы не два обстоятельства.

1. Предопределенный обработчик `EAssertionFailed` устроен таким образом, что выдает не шестнадцатеричный адрес ошибки, а имя файла с исходным текстом и номер строки, где произошла ИС, как показано на рис. 5.4.

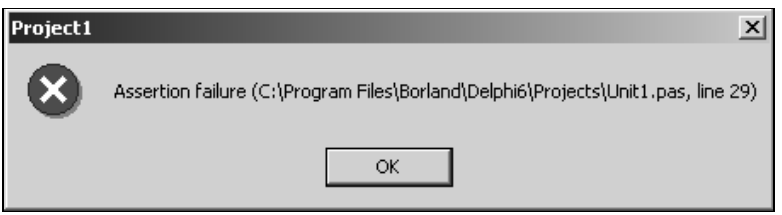


Рис. 5.4. Окно сообщения обработчика исключительной ситуации `EAssertionFailed`

2. При помощи специальной директивы компилятора `{$ASSERTIONS ON/OFF}` (или, что то же самое, `{$C+}/{$C-}`) возникновение этих ИС можно централизованно запретить. То есть в отлаживаемом коде в режиме `{$C+}` можно расставить вызов `Assert` во всех сомнительных и проверяемых местах. Когда же придет время генерации конечного варианта кода, переключением директивы на `{$C-}` весь отладочный вывод запрещается.

Резюме

Любое созданное в Delphi приложение должно обеспечивать обработку возможных исключительных ситуаций. Пусть вначале вам покажется ненужным создавать дополнительный код обработки ошибок для простейших программ, в которых вероятность ошибки минимальна. Зато впоследствии приобретенные навыки позволят вам существенно повысить надежность реальных приложений.

В Delphi для обработки исключительных ситуаций используются специальные конструкции языка Delphi и классы на основе базового класса исключительной ситуации `Exception`.

ЧАСТЬ II

Интерфейс и логика приложения

Глава 6

Приложение и проект

Глава посвящена наиболее общим вопросам создания приложений в среде разработки Delphi.

С организационной точки зрения основа разработки приложения — проект. Он объединяет все части приложения и обеспечивает их взаимодействие друг с другом и со средой разработки. Несколько проектов, предназначенных для создания самостоятельных частей одного сложного приложения, можно объединить в группе проектов.

Для поддержки общего функционала приложения в состав проекта включается специализированный файл проекта, который содержит очень важный код, осуществляющий инициализацию экземпляра класса приложения и его непосредственный запуск.

С программной точки зрения основой приложения является специальный класс `TApplication`, который автоматически включается в состав каждого проекта.

С точки зрения организации взаимодействия приложения с пользователем, существует несколько схем построения программного интерфейса, которые мы также рассмотрим здесь.

В этой главе рассматриваются следующие вопросы.

- Состав проекта.
- Группа проектов.
- Класс приложения `TApplication`.

Проект как основа разработки приложения

Разработка нового приложения начинается с создания проекта. *Проект* объединяет исходные файлы приложения в единое целое и обеспечивает их взаимодействие друг с другом и со средой разработки. Стандартный проект сразу после создания содержит три файла. Помимо файла проекта с расширением dpr он содержит исходные файлы форм и модулей.

Проект представляет ваше приложение на этапе разработки независимо от того, что именно вы хотите создать — исполняемый файл, динамическую библиотеку или объект COM или ActiveX.

Файл проекта содержит код, обеспечивающий функционирование проекта в среде разработки, создание и запуск приложения Windows. Он состоит из двух частей: секции `uses` и основного блока `begin...end` (листинг 6.1).

Листинг 6.1. Файл проекта

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form2};

{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

В секции `uses` перечислены все модули, входящие в состав проекта. В блоке `begin...end` после инициализации объекта приложения `TApplication` осуществляется создание форм, входящих в состав проекта. Метод `Run` обеспечивает запуск приложения.

Файл исходного кода модуля проекта имеет расширение pas. Он обеспечивает хранение описания связанной с модулем формы в синтаксисе Delphi, а также программной логики (листинг 6.2).

Листинг 6.2. Пример файла модуля проекта

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs;
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
```

```
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
const MessageStr = 'Щелчок на кнопке';
procedure TForm1.Button1Click(Sender: TObject);
begin
 ShowMessage(MessageStr);
end;
end.
```

Этот файл имеет обычную для модулей структуру. В секции `interface` содержится объявление классов, типов, констант и переменных. В секции `implementation` хранится исходный код процедур и функций. Напомним, что объявления, сделанные в секции `interface`, видны и в других модулях, а код секции `implementaion` доступен только в данном модуле. Директива `{$R *.DFM}` означает, что ресурсы для создания формы хранятся в файле с именем, аналогичным имени модуля, и расширением dfm.

В общем случае модуль может не иметь связанной с ним формы. В таких файлах обычно содержится исходный код динамических библиотек, общих для всего проекта процедур и функций.

Файл формы имеет расширение dfm и содержит описание свойств тех компонентов, которые размещены на форме в среде разработки (листинг 6.3).

Листинг 6.3. Пример файла формы проекта

```
object Form1: TForm1
  Left = 192
  Top = 107
  Width = 355
  Height = 236
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
```

```
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 128
    Top = 80
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
    OnClick = Button1Click
  end
end
```

Файл формы содержит описание только тех свойств компонентов, от которых зависят их внешний вид и положение на форме. Форма, описание которой представлено в листинге 6.3, показана на рис. 6.1.

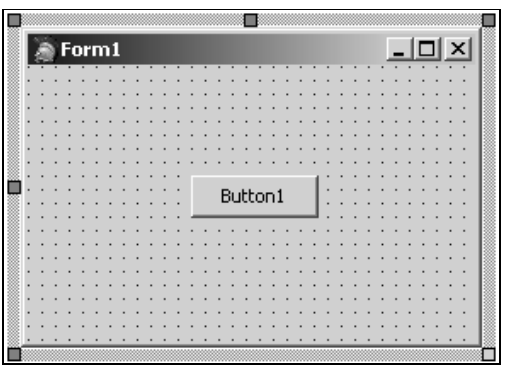


Рис. 6.1. Форма проекта

Примечание

Получить текстовое представление формы проекта можно при помощи команды **View As Text** из всплывающего меню формы.

Компиляция проекта осуществляется в соответствии со списком модулей в файле проекта (см. листинг 6.1). Откомпилированный модуль записывается в файл с расширением dcu. Если в исходный текст модуля были внесены изменения, то при компиляции проекта этот модуль перекомпилируется автоматически. В противном случае готовый код модуля добавляется к исполняемому файлу приложения (или DLL) из файла DCU.

При внесении изменений в исходный код файлов модуля, проекта или формы старый вариант сохраняется в файле с расширением ~dp, ~pa, ~df соответственно.

В проекте Delphi помимо модулей и форм могут присутствовать различные служебные файлы. Среди них важное место занимают ресурсы (расширение res), которые хранят изображения, строки для сообщений, информацию о версии программы и т. д. А в файле с расширением bdsproj хранятся настройки проекта и среды разработки для проекта.

Помимо проекта в Delphi присутствует и более сложная структурная единица — *группа проектов*, которая, как следует из названия, объединяет для совместной разработки несколько проектов. Например, было бы логично объединить в одной группе исполняемое главное приложение и динамическую библиотеку, в которой собраны все функции обработки или визуализации данных. Или серверную и клиентскую части двухуровневого приложения для работы с базами данных (БД).

Файл группы проектов имеет расширение bdsgroup и представляет собой текстовый файл с системной информацией и описанием проектов, входящих в группу (листинг 6.4).

Листинг 6.4. Пример файла группы проектов

```
п>ї<?xml version="1.0" encoding="utf-8"?>
<BorlandProject>
  <PersonalityInfo>
    <Option>
      <Option Name="Personality">Default.Personality</Option>
      <Option Name="ProjectType"></Option>
      <Option Name="Version">1.0</Option>
      <Option Name="GUID">{780DB670-DEE9-48C6-B40F-2F404A66EB3D}</Option>
    </Option>
  </PersonalityInfo>
  <Default.Personality>
      <Projects>
      <Projects Name="Project1.exe">Project1.bdsproj</Projects>
```

```
      <Projects Name="Targets">Project1.exe</Projects>
    </Projects>
    <Dependencies/>
  </Default.Personality>
</BorlandProject>
```

Строго говоря, единичный проект на самом деле является группой проектов, содержащей всего один элемент. В этом легко убедиться, заглянув в Менеджер проекта: даже в только что созданном проекте на первом месте списка находится группа, по умолчанию имеющая название `ProjectGroup1`. Правда, если возможности группы проекта не нужны, то и файл группы BDSGROUP автоматически не сохраняется.

Для управления группой проектов и проектами в среде разработки Delphi предназначен специальный инструмент — Менеджер проектов. Эта полезная утилита представляет собой окно с небольшой панелью инструментов и иерархическим раскрывающимся списком, который содержит в себе все составные части проекта или группы проектов (рис. 6.2).

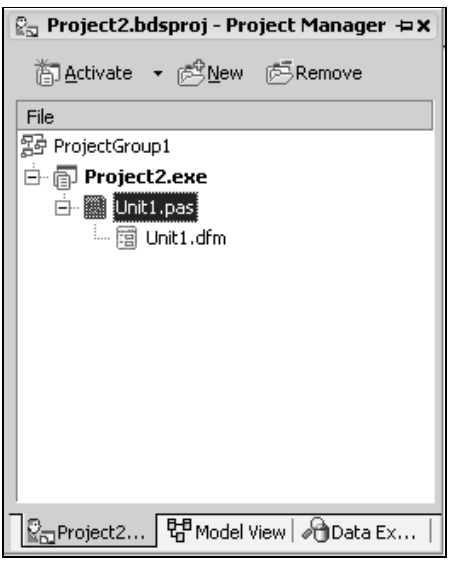


Рис. 6.2. Окно менеджера проектов

В группе можно добавить или удалить новый проект, в проекте — добавить или удалить новый модуль. Работа с группой проектов практически не отличается от работы с отдельным проектом, необходимо только следить за принадлежностью отдельных форм и модулей и вовремя переключаться на

нужный проект в Менеджере проектов. При компиляции в группе также нужно выбрать необходимый проект из списка в панели инструментов Delphi или Менеджера проектов.

Управление элементами осуществляется кнопками панели инструментов или командами из всплывающего меню.

Для выбора проекта предназначена кнопка **Activate** в панели инструментов Менеджера проектов.

Примечание

Возможность объединения групп или включения групп в другие группы отсутствуют.

Параметры проекта в целом доступны в диалоговом окне, которое открывается при выборе команды **Options** из всплывающего меню проекта в списке группы. В нем можно настроить компилятор и компоновщик, задать способ создания форм проекта, подключить файл помощи и значок.

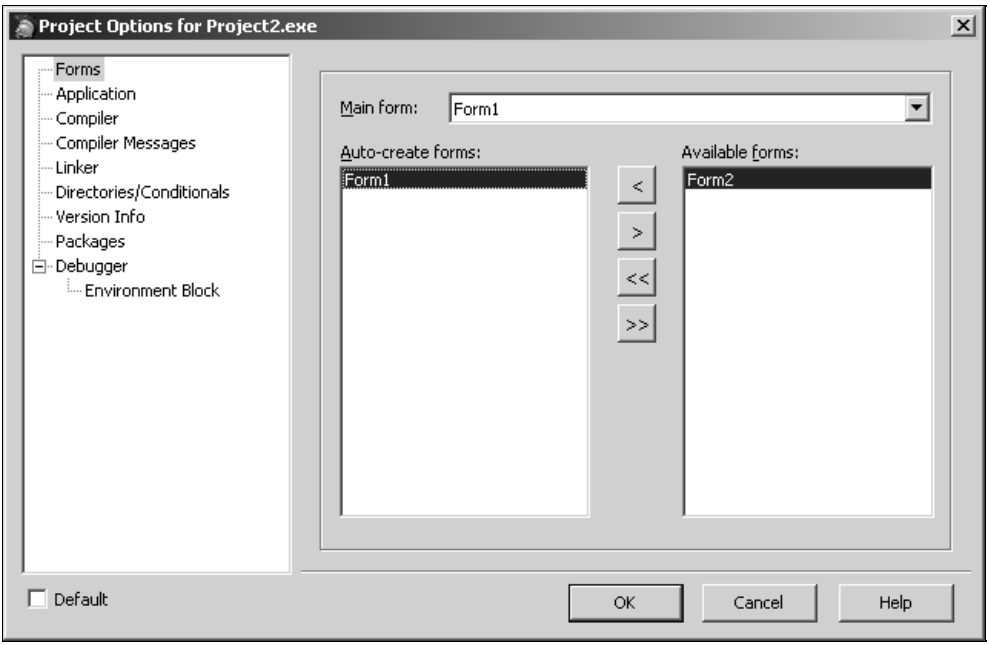


Рис. 6.3. Диалог параметров проекта, открытый на странице управления формами

Рассмотрим, например, управление формами проекта (рис. 6.3). При добавлении в проект новой формы ей присваивается статус создаваемой по умол-

чанию (список **Auto-create forms**). Это означает, что в файл проекта добавляется конструктор для этой формы, который вызывается при инициализации приложения (см. листинг 6.1). Следовательно, при запуске приложения создается и форма — ей отводятся необходимые ресурсы (дескриптор окна, область памяти и т. д.).

Такой механизм является достаточно универсальным и применяется в большинстве проектов. Однако если ваш проект содержит очень много форм, по запуск приложения может занимать достаточно продолжительное время. В этом случае те формы, которые используются редко или являются вспомогательными, при запуске приложения можно не создавать.

Для этого в диалоге параметров проекта достаточно переместить форму в список **Available forms** (см. рис. 6.3).

В результате из файла проекта удаляется конструктор соответствующей формы и приложение загружается быстрее. Но тогда обязанность создания формы ложится на программиста. Для этого достаточно поместить конструктор перед первым обращением к форме:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1 := TForm1.Create(Form1);
  Form1.ShowModal;
end;
```

Репозиторий Delphi

Delphi позволяет разработчику создавать самые разнообразные программы. При этом их структура может довольно сильно различаться, требуя дополнительных настроек проекта и применения специализированных объектов.

Заготовки (шаблоны) проектов и их составных частей содержатся в специальном хранилище — Репозитории (рис. 6.4). Основная цель создания Репозитория — предоставить разработчику возможности использования стандартных или уже готовых и отлаженных частей программы. Поэтому здесь можно найти готовое стандартное приложение, динамическую библиотеку, форму, модуль и т. п.

Программист может настроить Репозиторий в соответствии с собственными потребностями: удалить ненужные шаблоны и добавить собственные.

Открыть Репозиторий можно при помощи команды **File | New | Other** главного меню. Он представляет собой многостраничный блокнот, на страницах которого сгруппированы значки шаблонов проектов.

Состав объектов Репозитория можно обновлять, также можно создавать, переименовывать страницы и изменять состав объектов для них. Редактирова-

ние Репозитория осуществляется при помощи диалогового окна **Object Repository**, которое появляется при выборе команды **Repository** из меню **Tools** или команды **Repository Properties** из всплывающего меню Репозитория. Добавить новую форму в Репозиторий можно командой **Add to repository** из меню **Project**, командой **Add to repository** из всплывающего меню формы или перетаскиванием формы в окно Репозитория, в котором необходимо задать параметры будущего компонента.

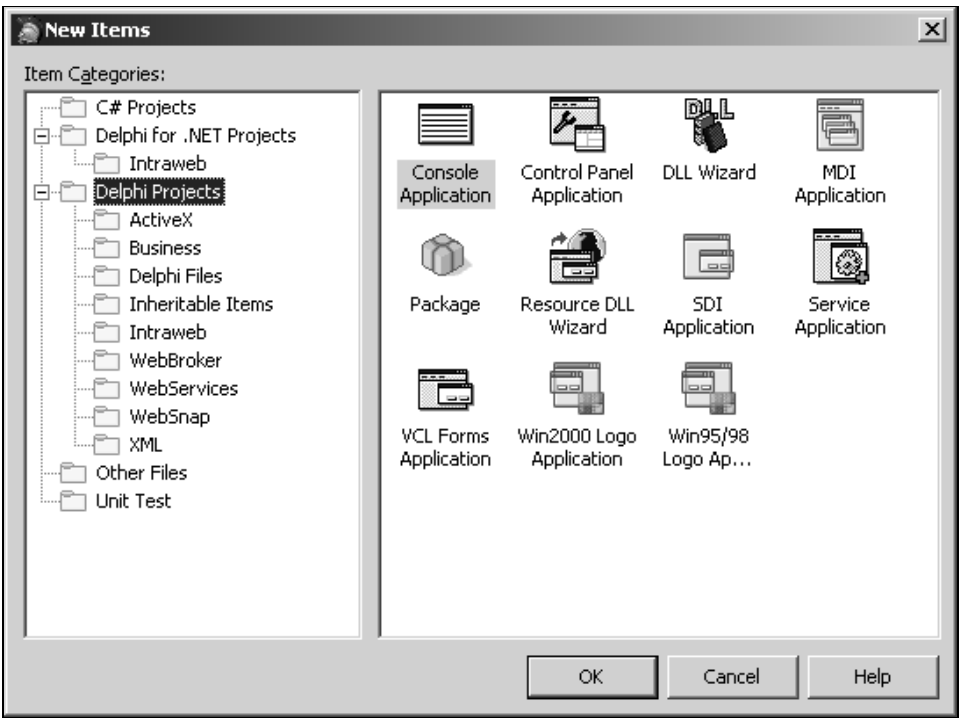


Рис. 6.4. Репозиторий Delphi

Класс *TApplication*

Приложение, работающее в операционной системе Windows, должно уметь выполнять ряд важных операций. Это обработка очереди сообщений, использование системных ресурсов, взаимодействие с файловой системой и т. д.

В библиотеке визуальных компонентов Delphi свойства и функции приложения Windows реализованы в классе `TApplication`, который является прямым потомком класса `TComponent`.

Экземпляр класса `TApplication` создается автоматически при помощи соответствующего конструктора при запуске исполняемого файла приложения. Ключевые методы класса обеспечивают создание, инициализацию, выполнение и уничтожение приложения. Затем экземпляр класса приложения доступен через переменную

```
var Application: TApplication;
```

Пример использования этой переменной имеется в каждом проекте приложения и представлен в листинге 6.1. В файле проекта стандартного приложения переменная `Application` служит для вызова следующих методов:

- `Initialize` — для инициализации приложения;
- `CreateForm` — для создания форм приложения;
- `Run` — для запуска приложения. В нем включается обработчик очереди сообщений метода `HandleMessage`, который выполняется до момента завершения работы приложения.

Весь этот код, жизненно важный для обеспечения работоспособности приложения, создается автоматически. Тем не менее разработчик может изменять код файла проекта по своему усмотрению. При этом доступны свойства и методы класса `TApplication`, описание которых приведено в табл. 6.1.

Таблица 6.1. *Свойства и методы класса* `TApplication`

Объявление	Описание
Свойства	
`property Active: Boolean;`	При значении `True` приложение активно и имеет фокус ввода
`proprety BiDiMode: TBiDiMode;`	Применяется для локализации приложения в арабских странах. Определяет направление чтения и выравнивания текстов
`property CurrentHelpFile: string;`	Содержит имя файла контекстной помощи, используемого приложением
`property DialogHandle: HWnd;`	Это свойство предназначено для хранения дескриптора окна диалога, созданного при помощи системной функции `CreateDialog`. Это бывает необходимо при использовании сообщений от диалогов других приложений
`property ExeName: string;`	Содержит имя исполняемого файла приложения

Таблица 6.1 *(продолжение)*

Объявление	Описание
Свойства	
`property Handle: HWND;`	Содержит дескриптор окна главной формы приложения. Значение присваивается операционной системой при создании приложения
`property HelpFile: string;`	Позволяет задать имя файла помощи для приложения
`property Hint: string;`	Содержит текст для ярлыка подсказки. Это свойство перекрывается аналогичными в элементах управления приложения
`property HintColor: TColor;`	Определяет цвет ярлыка подсказки для приложения
`property HintHidePause: Integer;`	Определяет время показа ярлыка подсказки в миллисекундах
`property HintPause: Integer;`	Определяет время до появления ярлыка подсказки
`property HintShortCuts: Boolean;`	Включает или отключает показ в тексте ярлыка подсказки для команд меню и т. п. горячих клавиш
`property HintShortPause: Integer;`	Определяет временной интервал при смене ярлыков при перемещении указателя по нескольким элементам управления
`property Icon: TIcon;`	Содержит значок приложения
`function IsRightToLeft: Boolean;`	Задает способ выравнивания элементов управления справа налево
`property MainForm: TForm;`	Возвращает указатель на экземпляр главной формы приложения
`property ShowHint: Boolean;`	Включает или отключает ярлыки подсказки во всем приложении
`property ShowMainForm: Boolean;`	Включает или отключает показ главной формы приложения при его запуске
`property Terminated: Boolean;`	Значение `True` сигнализирует о том, что приложение получило команду завершения
`property Title: string;`	Содержит текст под значком свернутого приложения

Таблица 6.1 *(продолжение)*

Объявление	Описание
Свойства	
`property UpdateFormatSettings: Boolean;`	При значении `True` изменения в системных установках форматирования автоматически срабатывают и в приложении
`property UpdateMetricSettings: Boolean;`	При значении `True` изменения в системных установках шрифтов ярлыков подсказки и текста под значком свернутого приложения автоматически срабатывают и в приложении
Методы	
`procedure BringToFront;`	Помещает активное окно приложения поверх любых других окон
`procedure CreateForm(FormClass: TFormClass; var Reference);`	Создает новую форму на основе класса, переданного в параметре `FormClass`. Указатель на созданную форму передается в переменную `Reference`. Владельцем формы становится приложение
`procedure CreateHandle;`	Создает главное окно приложения. Эта важная процедура необходима при запуске приложения. Разработчик не должен использовать ее напрямую
`function ExecuteAction(Action: TBasicAction): Boolean; reintroduce;`	Вызывает метод-обработчик `OnActionExecute`
`procedure HandleException(Sender: TObject);`	Обеспечивает обработку исключительных ситуаций, передавая управление соответствующему ошибке методу-обработчику
`procedure HandleMessage;`	Прерывает выполнение приложения и обеспечивает обработку очереди сообщений перед возвратом управления приложению
`function HelpCommand(Command: Word; Data: Longint): Boolean;`	Обеспечивает доступ к функциям Help API
`type THelpContext = -MaxLongInt..MaxLongInt;` `function HelpContext(Context: THelpContext): Boolean;`	Обеспечивает показ темы подсказки, идентификатор которой задан параметром `Context` из файла помощи, определенного свойством `HelpFile`

Таблица 6.1 *(продолжение)*

Объявление	Описание
Методы	
`function HelpJump(const JumpID: string): Boolean;`	Вызывает метод-обработчик `OnHelp`, который обеспечивает показ файла помощи. Параметр `JumpID` задает идентификатор темы помощи
`procedure HideHint;`	Убирает текущий ярлык подсказки
`type TWindowHook = function(var Message: TMessage): Boolean of object;` `procedure HookMainWindow(Hook: TWindowHook);`	Позволяет осуществлять перехват внешних диалоговых окон; заставляет их функционировать как обычное дочернее окно приложения. Параметр `Hook` определяет метод, обеспечивающий обработку сообщений для диалогового окна в главном окне приложения
`procedure Initalize;`	Осуществляет инициализацию приложения
`function MessageBox(Text, Caption: PChar; Flags: Longint): Integer;`	Выводит на экран окно сообщения. Параметр `Text` определяет текст сообщения. Параметр `Caption` задает заголовок окна. Параметр `Flags` задает стиль окна сообщения. Функция возвращает реакцию пользователя (щелчок на определенной кнопке)
`procedure Minimize;`	Сворачивает приложение
`procedure NormalizeAllTopMosts;`	Заставляет все окна, которые имеют стиль `fsStayOnTop` (располагаются поверх других окон), вести себя как обычные окна
`procedure NormalizeTopMosts;`	Заставляет все окна (кроме главного окна), которые имеют стиль `fsStayOnTop` (располагаются поверх других окон), вести себя как обычные окна
`procedure ProcessMessages;`	Прерывает выполнение операций в приложении для обработки сообщений от операционной системы
`procedure Restore;`	Восстанавливает свернутое приложение до обычных размеров
`procedure RestoreTopMosts;`	Восстанавливает состояние окон, измененное методом `NormalizeTopMosts`
`procedure Run;`	Запускает приложение
`procedure ShowException(E: Exception);`	Отображает диалог исключительной ситуации, заданной параметром `E`
`procedure Terminate;`	Завершает выполнение приложения

Таблица 6.1 *(продолжение)*

Объявление	Описание
Методы	
`type TWindowHook = function(var Message: TMessage): Boolean of object;` `procedure UnhookMainWindow(Hook: TWindowHook);`	Отменяет действие метода `HookMainWindow` для процедуры, заданной параметром `Hook`
`function UpdateAction(Action: TBasicAction): Boolean; reintroduce;`	Вызывает метод-обработчик `OnActionUpdate`
`function UseRightToLeftAlignment: Boolean;`	Возвращает `True`, если выравнивание текста в приложении находится в режиме справа налево (см. свойство `BiDiMode`)
`function UseRightToLeftReading: Boolean;`	Возвращает `True`, если свойство `BiDiMode` в приложении включено
`function UseRightToLeftScrollBar: Boolean;`	Возвращает `True`, если полоса вертикальной прокрутки в элементах управления приложения находится слева (см. свойство `BiDiMode`)
Методы-обработчики событий	
`TActionEvent = procedure (Action: TBasicAction; var Handled: Boolean) of object;` `property OnActionExecute: TActionEvent;`	Вызывается при выполнении любого действия в приложении *(см. главу 9)*. Параметр `Action` содержит указатель на объект действия. Параметр `Handled` возвращает `True`, если метод-обработчик связан с действием
`TActionEvent = procedure (Action: TBasicAction; var Handled: Boolean) of object;` `property OnActionUpdate: TActionEvent;`	Вызывается при выполнении метода `Update` любого действия приложения *(см. главу 9)*. Параметр `Action` содержит указатель на объект действия. Параметр `Handled` возвращает `True`, если метод-обработчик связан с действием и выполняет его обработку
`property OnActivate: TNotifyEvent;`	Вызывается при активизации приложения

Таблица 6.1 *(продолжение)*

Объявление	Описание
Методы-обработчики событий	
`property OnDeactivate: TNotifyEvent;`	Вызывается при деактивизации (сворачивании, переключении на другое приложение) приложения
`type TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;` `property OnException: TExceptionEvent;`	Вызывается при возникновении исключительной ситуации, обработка которой не предусмотрена в приложении. Параметр `Sender` определяет объект, вызвавший исключительную ситуацию. Параметр `E` ссылается на экземпляр класса исключительной ситуации
`type THelpEvent = function (Command: Word; Data: Longint; var CallHelp: Boolean): Boolean of object;` `property OnHelp: THelpEvent;`	Вызывается при обращении к системе помощи из приложения. Параметр `CallHelp` управляет вызовом системы помощи после выполнения обработчика. При значении `True` система помощи вызывается
`property OnHint: TNotifyEvent;`	Вызывается при перемещении указателя мыши по элементу управления приложения, который может отображать ярлык подсказки
`type TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;` `property OnIdle: TIdleEvent`	Вызывается при переходе приложения в режим ожидания (при прекращении выполнения каких-либо действий). Параметр `Done` управляет поведением приложения после выполнения обработчика. При значении `True` следом за обработчиком вызывается системная функция `WaitMessage`, которая не передает управление другим приложениям, пока в данном приложении не начнется выполнение операций
`type TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of object;` `property OnMessage: TMessageEvent;`	Вызывается при получении сообщения от операционной системы. Параметр `Msg` определяет объект сообщения. Параметр `Handled` управляет обработкой сообщения. При значении `True` осуществляется стандартная обработка

Таблица 6.1 *(окончание)*

Объявление	Описание
Методы-обработчики событий	
`type TNotifyEvent = procedure (Sender: TObject) of object;` `property OnMinimize: TNotifyEvent;`	Вызывается при сворачивании приложения
`type TNotifyEvent = procedure (Sender: TObject) of object;` `property OnRestore: TNotifyEvent;`	Вызывается при восстановлении свернутого приложения

`type`
`TWMKey = record`
`Msg: Cardinal;`
`CharCode: Word;`
`Unused: Word;`
`KeyData: Longint;`
`Result: Longint;`
`end;`
`TShortCutEvent = procedure (var Msg: TWMKey; var Handled: Boolean) of object;`
`property OnShortCut: TShortCutEvent;`
	Вызывается при нажатии клавиши в активном приложении.
	Параметр `Msg` определяет параметры нажатой клавиши.
	Параметр `Handled` управляет обработкой нажатия. При значении `True` осуществляется обработка стандартными средствами

`TShowHintEvent = procedure (var HintStr: string; var CanShow: Boolean; var HintInfo: ThintInfo) of object;` `property OnShowHint: TShowHintEvent;`	Вызывается при показе ярлыка подсказки. Параметр `HintStr` содержит текст ярлыка. Параметр `CanShow` управляет видимостью ярлыка. Параметр `HintInfo` содержит информацию о ярлыке

Атрибуты приложения

Приложение, функционирующее в операционной системе Windows, должно удовлетворять ряду требований. Помимо разнообразных программных ас-

пектов, такое приложение должно иметь определенный набор свойств, которые используются на различных этапах работы программы. Эти свойства реализованы в классе `TApplication`.

Дескриптор главного окна приложения (этот указатель играет важную роль для многих системных функций) содержится в свойстве

```
property Handle: HWND;
```

Объектный тип, указывающий на экземпляр класса главной формы приложения, представлен в свойстве

```
property MainForm: TForm;
```

Значок для представления свернутого приложения, отображающийся в заголовке главного окна программы, содержится в свойстве

```
property Icon: TIcon;
```

Свойство

```
property Title: string;
```

представляет текст, который отображается под значком свернутого приложения.

Для сворачивания приложения достаточно вызвать метод `Minimize`, для восстановления исходного состояния — `Restore`.

Обработка сообщений

В течение одной секунды приложение может получать и обрабатывать несколько тысяч сообщений. Для их обработки в каждом приложении Windows предусмотрен специальный механизм. В подавляющем большинстве случаев у разработчика нет необходимости что-либо изменять в нем. Однако управление обработкой сообщений на более высоком уровне встречается довольно часто.

Если процесс выполнения операции занимает длительное время, то разработчик должен предусмотреть вывод информации о состоянии процесса и возможность его прерывания. Для этого можно использовать метод `ProcessMessages`, который осуществляет обработку входной очереди сообщений.

Рассмотрим небольшой пример (рис. 6.5). Пусть при щелчке на кнопках **`Btn1`** и **`Btn2`** осуществляется расчет таблицы умножения от 1 до 100. В процессе расчета его текущее состояние должно отображаться на индикаторе `ProgressBar`. Кроме этого, пользователь должен иметь возможность прервать выполнение операции.

Рис. 6.5. Главная форма проекта DemoProcessMsg

Исходный код, работающий при щелчке на кнопке **Btn1**, не сможет выполнить поставленные условия в полном объеме, так как цикл обработки данных не передает управление до момента окончания работы.

Примечание

Компонент `TProgressBar` разработан таким образом, что каждый вызов метода `StepIt` не только изменяет его состояние, но и обеспечивает отрисовку компонента. Поэтому даже в циклах компонент успешно показывает ход выполнения операции.

Исходный код, работающий при щелчке на кнопке **Btn2**, после каждого умножения передает управление методу `ProcessMessages`, который осуществляет обработку входной очереди сообщений для приложения. Это позволяет не только обеспечить возможность прерывания операции. Листинг 6.5 содержит код секции `implementation` модуля главной формы проекта DemoProcessMsg.

Листинг 6.5. Секция `implementation` модуля главной формы проекта DemoProcessMsg

```
implementation
{$R *.DFM}
procedure TMainForm.Btn1Click(Sender: TObject);
var i, j, x : Integer;
begin
 ProgressBar.Position := 0;
 for i := 1 to 100 do
 for j := 1 to 100 do
```

```
 begin
  x := i * j;
  ProgressBar.StepIt;
  if CheckBox.Checked then Abort;
 end;
end;
procedure TMainForm.Btn2Click(Sender: TObject);
var i, j, x : Integer;
begin
 ProgressBar.Position := 0;
 for i := 1 to 100 do
 for j := 1 to 100 do
 begin
  x := i * j;
  Application.ProcessMessages;
  ProgressBar.StepIt;
  if CheckBox.Checked then Abort;
 end;
end;
procedure TMainForm.Btn3Click(Sender: TObject);
var i, j, x : Integer;
begin
 ProgressBar.Position := 0;
 for i := 1 to 100 do
 for j := 1 to 100 do
 begin
  x := i * j;
  Application.HandleMessage;
  ProgressBar.StepIt;
  if CheckBox.Checked then Abort;
 end;
end;
end.
```

Попробуйте, например, во время расчета переместить над окном приложения любое другое окно. При выполнении процесса кнопки **Btn1** приложение не сможет правильно отображать себя до момента окончания вычислений, т. к. все системные сообщения о необходимости перерисовки формы останутся во входной очереди до окончания расчетов. При выполнении

процесса кнопки **Btn2** приложение за счет метода `ProcessMessages` имеет возможность регулярно просматривать очередь и адекватно реагировать на сообщения операционной системы.

Метод `HandleMessage` обрабатывает всего одно сообщение из очереди и затем передает управление обратно. Если входная очередь пуста, то управление передается методу-обработчику `OnIdle` (если он определен). Поэтому обработка входной очереди этим методом может быть существенно замедлена.

В нашем примере обработка очереди сообщений этим методом осуществляется при вычислениях в методе-обработчике кнопки **Btn3**. После щелчка на этой кнопке приращение индикатора будет осуществляться только при передаче сообщений в очередь приложения. Например, очередь можно загрузить сообщениями о перерисовке, для этого достаточно постоянно перемещать указатель мыши по форме приложения.

На получение сообщения в приложении реагирует метод-обработчик

```
type TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of
object;
property OnMessage: TMessageEvent;
```

в котором можно реализовать специфическую реакцию приложения на то или иное сообщение. Идентификация сообщения производится при помощи параметра `Msg`:

```
TMsg = packed record
    hwnd: HWND;
    message: UINT;
    wParam: WPARAM;
    lParam: LPARAM;
    time: DWORD;
    pt: TPoint;
end;
```

Идентификатор сообщения содержится в свойстве `message` типа `TMsg`.

Примечание

Запись `TMsg` является упакованной (зарезервированное слово `packed`). Подробнее об этом *см. главу 2*.

Предупреждение

Методом `OnMessage` следует пользоваться крайне осторожно, т. к. этот код будет работать при получении любого сообщения, а их в одну секунду может поступить несколько тысяч.

Реакция на действия пользователей

Большинство визуальных компонентов Delphi предоставляют разработчику богатый набор средств отслеживания действий пользователя. Самыми распространенными из них являются обработчики манипуляций с мышью и клавиатурой.

В классе `TApplication` имеется еще один метод-обработчик `OnShortCut`, который реагирует на нажатие клавиши. Его ценность заключается в том, что он вызывается самым первым — даже до метода-обработчика `OnKeyDown` элемента управления.

```
TShortCutEvent = procedure (var Msg: TWMKey; var Handled: Boolean) of
object;
property OnShortCut: TShortCutEvent;
```

Информация о нажатой клавише содержится в параметре `Msg`:

```
TWMKey = record
    Msg: Cardinal;
    CharCode: Word;
    Unused: Word;
    KeyData: Longint;
    Result: Longint;
  end;
```

Код клавиши содержится в свойстве `CharCode`.

Система помощи

Класс `TApplication` позволяет организовать управление системой помощи на уровне всего приложения.

Свойство

```
property HelpFile: string;
```

позволяет задать имя справочного файла, который будет использоваться всеми элементами управления во всех формах приложения. Если с каждым визуальным компонентом связать собственную тему контекстной помощи *(см. главу 8)*, то при ее вызове для определения имени файла подсказки будет использовано это свойство. Если свойство `HelpFile` не задать, то помощь в таком приложении просто не будет работать.

В необходимых случаях вы можете вызвать нужную тему контекстной подсказки при помощи метода `HelpContext`. Он удобен, когда содержание помощи должно изменяться в зависимости от текущей ситуации:

```
const SomeTopic1 = 41;
      SomeTopic2 = 42;
...
```

```
if CheckBox1.Checked
  then Application.HelpContext(SomeTopic1)
  else Application.HelpContext(SomeTopic2);
```

В параметре метода задается идентификатор темы помощи.

При обращении к системе помощи из приложения вызывается метод-обработчик `OnHelp`, в котором можно предусмотреть выполнение необходимых дополнительных операций. Для этого в какой-либо форме создается метод, параметры которого соответствуют методу-обработчику:

```
function TForm1.HelpFileControl(Command: Word; Data: Longint;
                                var CallHelp: Boolean): Boolean;
begin
 if Application.HelpFile = '' then
 if HelpFileDialog.Execute then
 Application.HelpFile := HelpFileDialog.FileName;
end;
```

Здесь осуществляется проверка имени файла помощи для приложения и в случае его отсутствия вызывается диалог выбора файла *(см. главу 7)*. Затем в методе-обработчике `OnCreate` этой формы переменной процедурного типа класса приложения присваивается созданный метод:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Application.OnHelp := HelpFileControl;
end;
```

Эту же операцию можно выполнить и в файле проекта, но обязательно после создания формы:

```
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.OnHelp := Form1.HelpFileControl;
Application.Run;
```

Резюме

В этой главе рассмотрена организация проекта Delphi и программная реализация класса приложения.

Проект объединяет части приложения, обеспечивает его разработку и взаимодействие программиста со средой Delphi. В Репозитории Delphi хранятся различные типы наиболее часто используемых проектов. При необходимости состав проектов Репозитория можно изменять. Здесь же можно выбрать тип интерфейса создаваемого приложения.

Выполнение важнейших функций приложения Windows обеспечивает класс `TApplication`, экземпляр которого имеется в любом проекте приложения.

Глава 7

Форма

Большинство приложений Windows обеспечивают взаимодействие с пользователем при помощи совокупности элементов управления, размещенных в одном или нескольких окнах. Ввод и отображение данных, управляющие воздействия пользователя, отображение состояния приложения и отдельных процессов, — все эти операции выполняются с активным использованием окон приложения.

Стандартный проект в Delphi обычно состоит исключительно из форм. Форма — это всего лишь окно приложения на этапе разработки. Каждую форму определяют два файла: файл формы, описывающий свойства формы и размещенных на ней компонентов, и модуль, содержащий описание экземпляра класса формы и исходный код *(см. главу 6)*.

Свойства и поведение окна инкапсулированы в очень важном классе `TForm`, который является отдаленным потомком класса `TWinControl`. Благодаря возможностям своих предков окно приложения Delphi не только обеспечивает работу пользовательского интерфейса, но и является контейнером для других компонентов, организуя управление ими на этапах разработки и выполнения.

Важность класса формы определяется двумя основными аспектами.

- С точки зрения создания исходного кода, класс формы объединяет все размещенные на ней компоненты и обеспечивает хранение программной логики, реализованной в процедурах и функциях.
- С точки зрения разработки приложения в целом, создание экземпляров классов форм на этапе загрузки программы обеспечивает работу пользовательского интерфейса.

В этой главе обсуждаются следующие вопросы.

- Почему окно приложения в среде разработки называется формой.
- Как взаимодействует форма и приложение.
- Возможности класса формы.

- Как использовать шаблоны форм.
- Что такое наследование форм.

Роль формы в приложении

Итак, приложение Windows должно иметь хотя бы одно окно. Даже если приложение при запуске ничего не показывает на экране, для него все равно автоматически создается дескриптор главного окна — указатель на область памяти, отведенную для его размещения. А для консольных приложений система всегда автоматически создает окно.

Окно приложения должно "уметь" выполнять целый ряд важных операций: правильно отображать себя на экране, взаимодействовать с другими окнами и операционной системой, управлять размещенными на нем элементами управления, реагировать на разнообразные события.

В среде разработки Delphi понятие окна приложения совмещено с понятием формы. По существу это одно и то же. Просто формой называется окно приложения на этапе разработки. Форме при разработке и окну при выполнении приложения соответствует один и тот же экземпляр класса; тем не менее введение понятие "форма" представляется оправданным.

Дело в том, что на этапе разработки окно должно обладать целым рядом ценных свойств, которые обеспечивают процесс программирования и обязаны исчезнуть при запуске приложения.

Самые важные из них заключается в том, что форма умеет взаимодействовать с инструментами среды разработки (Палитрой инструментов, Инспектором объектов и т. д.) и размещенными на ней компонентами. Первое из названных свойств обеспечивается возможностями, унаследованными от базового класса иерархии VCL `TComponent`. Второе свойство обеспечено механизмами самого класса формы.

Примечание

Для создания формы в среде разработки осуществляется вызов системной функции `WinMain`, в которую подставляются необходимые параметры.

Кроме этого, форма обеспечивает еще целый ряд удобных и полезных для разработчика функций. Это разметка рабочей области, возможность выравнивания компонентов и т. д.

Для хранения каждой формы проекта необходимы два файла. Файл с расширением dfm хранит описание свойств формы и размещенных на ней компонентов. Для хранения исходного кода класса формы и ее компонентов предназначен файл модуля с расширением pas.

Формы играют исключительно важную роль в процессе создания приложения. Они обеспечивают создание пользовательского интерфейса с одной стороны и хранение программной логики приложения — с другой. Хотя форма и является потомком класса `TComponent`, она не присутствует в Палитре инструментов. Новая форма проекта создается командой **New Form - Delphi for Win32** из меню **File** или из Репозитория. Это говорит о роли формы в процессе разработки приложения — на нее возложено гораздо больше обязанностей, чем на обычный компонент.

С точки зрения исходного кода, любой размещенный на форме компонент представляет собой всего лишь новую переменную объектного типа, объявленную в классе формы. Это объявление осуществляется автоматически в момент размещения компонента на форме. В результате в модуле этой формы сразу же становятся доступными все свойства и методы этих компонентов. Форма, на которой находится компонент, становится его владельцем. Пример модуля формы проекта иллюстрирует листинг 7.1.

Листинг 7.1. Модуль формы проекта

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Buttons;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    SpeedButton1: TSpeedButton;
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
uses Unit2;
```

```
procedure TForm1.FormShow(Sender: TObject);
begin
 Label1.Caption := '';
 Edit1.Clear;
end;
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 if SpeedButton1.Down
 then Form2.Label1.Caption := 'В главной форме нажата кнопка';
end;
end.
```

Для использования свойств и методов компонентов из модуля формы-владельца можно обращаться к этим компонентам по имени объектной переменной. Для обращения из других модулей необходимо добавлять имя формы-владельца. В листинге 7.1 при открытии формы в методе-обработчике `FormShow` идет прямое обращение к собственным компонентам, а при закрытии формы в методе-обработчике `FormClose` в другую форму передается текстовое сообщение.

Форма, как владелец размещенных на ней компонентов, позволяет эффективно управлять ими на этапе выполнения приложения. Для этого класс формы имеет ряд свойств и методов, которые позволяют получить доступ к любому компоненту. Например, программными средствами можно передавать фокус от одного элемента управления другому; выполнять для всех компонентов или их любого подмножества заданную последовательность действий.

Методы-обработчики формы также играют важную роль при создании приложений. Практика программирования показывает, что в каждой форме имеются операции, которые необходимо выполнить при создании, открытии формы и ее закрытии.

В зависимости от настроек свойств, формы в проекте могут играть разные роли. Форма может быть главной формой проекта. В этом случае она автоматически появляется при запуске приложения, а ее закрытие приводит к завершению работы всей программы.

Дочерние формы равноправны между собой. Их можно запускать в модальном и немодальном режиме. Форма в модальном режиме не позволяет передавать фокус управления элементам управления других форм приложения. Это можно сделать только после закрытия формы. Свободное переключение возможно между формами, работающими в немодальном режиме.

Формы в приложении различаются также по стилю оформления. Это может быть классическое окно с заголовком, рамкой, системными меню и кнопками или диалог и т. д.

На этапе выполнения форма активно взаимодействует с классом приложения, т. к. именно через него форма получает все системные сообщения и взаимодействует с другими формами и внешней средой. В этом процесс важную роль играет дескриптор окна, обеспечивающий его идентификацию на программном уровне. Дальше мы детально рассмотрим возможности, которые предоставляет разработчику класс формы `TForm`.

Класс *TForm*

Класс `TForm` инкапсулирует форму приложения Delphi. Помимо свойств стандартного окна, форма обладает дополнительными возможностями, позволяющими ей быть важнейшей частью процесса разработки приложения и хранилищем для других визуальных и невизуальных компонентов.

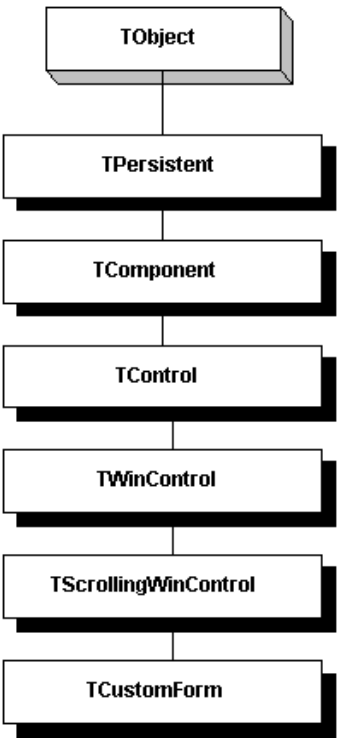


Рис. 7.1. Иерархия предков класса `Tform`

Эти функции класс формы частично наследует от многочисленных предков, частично реализует самостоятельно. Иерархия предков класса `TForm` представлена на рис. 7.1.

Приведем описание всех этих классов.

- `TComponent` — является той основой, которая позволяет форме быть платформой для других компонентов на этапе разработки приложения. Унаследованные от него механизмы обеспечивают взаимодействие формы со средой разработки, проектом и другими компонентами.
- `TControl` — обеспечивает функции визуализации экземпляра класса `TForm`. Благодаря ему форма умеет общаться с графической подсистемой Windows.
- `TWinControl` — придает своему потомку свойства оконного элемента управления. Благодаря его возможностям форма воспринимается операционной системой как полноценное окно.
- Класс `TScrollingWinControl` — обеспечивает форму механизмом горизонтальной и вертикальной прокрутки. Он используется, если какой-либо из элементов управления формы выходит за пределы ее рабочей области.
- `TCustomForm` — делает форму формой. От него стандартная форма наследует 90% своих свойств и методов.

Свойства и методы класса `TForm` приведены в табл. 7.1.

Таблица 7.1. *Свойства и методы класса* `TForm`

Объявление	Описание
Свойства	
`property Action: TBasicAction;`	Назначает для формы действие *(см. главу 9)*
`property Active: Boolean;`	Значение `True` сигнализирует о том, что форма имеет фокус
`property ActiveControl: TWinControl;`	Определяет элемент управления формы, который имеет фокус
`property ActiveMDIChild: TForm;`	Для главной формы MDI определяет активную дочернюю форму
`property ActiveOLEControl: TWinControl;`	Определяет активный элемент управления OLE на форме

Таблица 7.1 *(продолжение)*

Объявление	Описание
Свойства	

`type TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);`
`TBorderIcons = set of TBorderIcon;`
`property BorderIcons:`
`TBorderIcons;`
	Определяет комбинацию системных кнопок в заголовке формы. Значение этого свойства может войти в противоречие со свойством `BorderStyle`.
	`biSystemMenu` — кнопка системного меню.
	`biMinimize` — кнопка сворачивания.
	`biMaximize` — кнопка разворачивания.
	`biHelp` — кнопка помощи

`type TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog, bsToolWindow, bsSizeToolWin);` `property BorderStyle: TFormBorderStyle;`	Задает стиль рамки формы. `bsNone` — все элементы рамки отсутствуют. `bsSingle` — стандартная форма без изменения размеров. `bsSizeable` — стандартная форма с изменением размеров. `bsDialog` — рамка в стиле диалога. `bsToolWindow` — стиль панели инструментов без изменения размеров. `bsSizeToolWin` — стиль панели инструментов с изменением размеров
`property Canvas: TCanvas;`	Канва формы *(см. главу 12)*
`property ClientHandle: HWND;`	Содержит дескриптор активного дочернего окна MDI
`property ClientHeight: Integer;`	Высота клиентской области формы в пикселах
`property ClientRect: TRect;`	Определяет прямоугольник клиентской области формы в пикселах
`property ClientWidth: Integer;`	Ширина клиентской области в пикселах

`type`
`TDefaultMonitor = (dmDesktop, dmPrimary, dmMainForm, dmActiveForm);`
`property DefaultMonitor: TDefaultMonitor;`
	Определяет режим работы формы в многомониторных системах.
	`dmDesktop` — монитор не определяется.
	`dmPrimary` — форма появится на мониторе, который стоит первым в свойстве `Monitors` глобального объекта `Screen`.

Таблица 7.1 *(продолжение)*

Объявление	Описание
Свойства	
	`dmMainForm` — форма появляется на том же мониторе, что и главная форма. `dmActiveForm` — форма появляется на том же мониторе, что и текущая форма
`property DropTarget: Boolean;`	Значение `True` говорит о том, что форма является целью операции Drag And Drop
`property Floating: Boolean;`	Значение `True` говорит о том, что форма находится в доке другой формы
`type TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop);` `property FormStyle: TFormStyle;`	Определяет режим работы формы. `fsNormal` — форма SDI. `fsMDIChild` — дочерняя форма MDI. `fsMDIForm` — главная форма MDI. `fsStayOnTop` — форма всегда находится поверх других форм
`property HelpFile: string;`	Содержит имя файла помощи приложения
`property Icon: TIcon;`	Определяет для значок формы
`property KeyPreview: Boolean;`	При значении `True` методы-обработчики событий от клавиатуры сначала выполняются для формы, затем для активного элемента управления. При значении `False` эти методы-обработчики выполняются только для активного элемента управления
`property MDIChildCount: Integer;`	Содержит число дочерних форм MDI
`property MDIChildren[I: Integer]: TForm;`	Индексированный список указателей на дочерние формы MDI
`property Menu: TMainMenu;`	Определяет объект меню для формы *(см. главу 9)*

`type TModalResult = Low(Integer)..`
`High(Integer);`
`property ModalResult: TModalResult;`
	Содержит возвращаемое модальной формой значение

Таблица 7.1 *(продолжение)*

Объявление	Описание
Свойства	
`property ObjectMenuItem: TMenuItem;`	Определяет элемент меню, который изменяет свое состояние в результате работы в форме объекта OLE
`property OldCreateOrder: Boolean;`	При значении `True` методы-обработчики `OnCreate` и `OnDestroy` выполняются после всех конструкторов и деструкторов, иначе — до конструкторов и деструкторов
`property OleFormObject: IOleForm;`	Определяет интерфейс `IOleForm` для всех объектов OLE формы
`property Parent: TWinControl;`	Определяет оконный элемент управления — владелец формы
`property ParentBiDiMode: Boolean;`	При значении `True` в форме действует значение свойства `BiDiMode` объекта приложения
`property PixelsPerInch: Integer;`	Определяет число пикселов, приходящееся на один дюйм экрана
`type TPosition = (poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly, poScreenCenter, poDesktopCenter);` `property Position: TPosition;`	Задает положение формы на экране. `poDesigned` — размеры и положение формы соответствуют значениям во время разработки. `poDefault` — положение и размер формы определяются ОС. `poDefaultPosOnly` — размер соответствует размеру во время разработки, положение определяется ОС. `poDefaultSizeOnly` — положение соответствует положению во время разработки, размер определяется ОС. `poScreenCenter` — размер соответствует размеру во время разработки, форма находится в центре экрана. `poDesktopCenter` — размер соответствует размеру во время разработки, форма находится в центре экрана (для многомониторных систем)

Таблица 7.1 *(продолжение)*

Объявление	Описание
Свойства	
`type TPrintScale = (poNone, poProportional, poPrintToFit);` `property PrintScale: TPrintScale;`	Определяет масштаб формы при печати. `poNone` — не масштабируется. `poProportional` — масштаб подбирается так, чтобы отпечаток соответствовал размерам формы на экране. `poPrintToFit` — пропорции отпечатка соответствуют пропорциям на экране
`property Scaled: Boolean;`	При значении `True` при необходимости размеры формы приводятся в соответствие текущему значению `PixelsPerInch`
`type TTileMode = (tbHorizontal, tbVertical);` `property TileMode: TTileMode;`	Определяет расположение дочерних форм MDI при вызове метода `Tile`
`property Visible: Boolean;`	Определяет видимость приложения
`property WindowMenu: TMenuItem;`	Для главного окна MDI определяет стандартное меню Windows
`type TWindowState = (wsNormal, wsMinimized, wsMaximized);` `property WindowState: TWindowState;`	Возвращает текущее состояние формы
Методы	
`procedure AfterConstruction; override;`	Генерирует событие `OnCreate`. Для этого свойство `OldCreateOrder` должно иметь значение `False`
`procedure ArrangeIcons;`	Упорядочивает значки свернутых дочерних окон MDI
`procedure BeforeDestruction; override;`	Генерирует событие `OnDestroy`. Для этого свойство `OldCreateOrder` должно иметь значение `False`
`procedure Cascade;`	Располагает дочерние окна MDI каскадом
`procedure Close;`	Закрывает форму
`function CloseQuery: Boolean; virtual;`	Возвращает `True`, если форму можно закрыть

Таблица 7.1 *(продолжение)*

Объявление	Описание
Методы	
`procedure FocusControl(Control: TWinControl);`	Передает фокус элементу управления, определенному параметром `Control`
`function GetFormImage: TBitmap;`	Возвращает изображение формы
`procedure Hide;`	Убирает форму с экрана, не уничтожая ее

`type`
`TWMKey = record`
`  Msg: Cardinal;`
`  CharCode: Word;`
`  Unused: Word;`
`  KeyData: Longint;`
`  Result: Longint;`
`end;`
`function IsShortCut(var Message: TWMKey): Boolean; dynamic;`
	Вызывается автоматически при использовании клавиатуры. Возвращает `True`, если в активной форме была нажата комбинация горячих клавиш, определенных для этой формы. Параметр `Message` позволяет определить нажатую клавишу или комбинацию клавиш. Функция генерирует событие `OnShortCut`

`procedure MouseWheelHandler(var Message: TMessage); override;`	Вызывается автоматически при передаче в активную форму события перемещения мыши. Этот метод передает обработку события в активный элемент управления, вызывая его метод `Perform`
`procedure Next;`	Передает фокус следующей дочерней форме MDI
`procedure Previous;`	Передает фокус предыдущей дочерней форме MDI
`procedure Print; virtual;`	Печатает изображение формы
`procedure Release;`	Уничтожает форму и освобождает занимаемую память. Выполняется только после окончания работы всех методов-обработчиков
`procedure SendCancelMode(Sender: TControl);`	Прерывает обработку сообщений от мыши, прокрутки и т. д.
`procedure SetFocus;`	Передает форме фокус
`function SetFocusedControl(Control : TWinControl): Boolean; virtual;`	Передает фокус элементу управления, заданному параметром `Control`. Если элемент уже находится в процессе получения фокуса, возвращает `False`, во всех других случаях возвращает `True`

Таблица 7.1 *(продолжение)*

Объявление	Описание
Методы	
`procedure Show;`	Выводит форму на экран. Форма начинает работать в немодальном режиме
`function ShowModal: Integer; virtual;`	Выводит форму на экран. Форма начинает работать в модальном режиме
`procedure Tile;`	Располагает дочерние формы MDI мозаикой
`function WantChildKey(Child: TControl; var Message: TMessage): Boolean; virtual;`	Эту функцию вызывает любой элемент управления формы, который получает событие ввода с клавиатуры
Методы-обработчики событий	
`property OnActivate: TNotifyEvent;`	Вызывается при передаче форме фокуса

`type`
`TCloseAction = (caNone, caHide, caFree, caMinimize);`
`TCloseEvent = procedure(Sender: TObject; var Action: TCloseAction) of object;`
`property OnClose: TCloseEvent;`
	Вызывается при закрытии формы. Параметр `Action` определяет поведение формы при закрытии.
	`caNone` — форма не закрывается.
	`caHide` — форма становится невидимой.
	`caFree` — форма закрывается.
	`caMinimize` — форма сворачивается

`type TCloseQueryEvent = procedure(Sender: TObject; var CanClose: Boolean) of object;` `property OnCloseQuery: TCloseQueryEvent;`	Вызывается при закрытии формы. Специально предназначен для выполнения операций, позволяющих пользователю принять решение о закрытии формы или продолжении работы. Параметр `CanClose` определяет, закрывать форму или нет
`property OnCreate: TNotifyEvent;`	Вызывается при создании формы
`property OnDeactivate: TNotifyEvent;`	Вызывается при потере формой фокуса
`property OnDestroy: TNotifyEvent;`	Вызывается при уничтожении формы

Таблица 7.1 *(окончание)*

Объявление	Описание
Методы-обработчики событий	
`type THelpEvent = function (Command: Word; Data: Longint; var CallHelp: Boolean): Boolean of object;` `property OnHelp: THelpEvent;`	Вызывается при обращении из формы к системе помощи приложения
`property OnHide: TNotifyEvent;`	Вызывается при исчезновении формы с экрана
`property OnPaint: TNotifyEvent;`	Вызывается при перерисовке формы

`type`
`TWMKey = record`
`  Msg: Cardinal;`
`  CharCode: Word;`
`  Unused: Word;`
`  KeyData: Longint;`
`  Result: Longint;`
`end;`
`TShortCutEvent = procedure (var Msg: TWMKey; var Handled: Boolean) of object;`
`property OnShortCut: TShortCutEvent;`
	Вызывается при нажатии клавиши.
	Параметр `Msg` содержит информацию о событии и коде клавиши.
	Параметр `Handled` определяет место обработки сообщения. При значении `True` обработка проводится в методе-обработчике

`property OnShow: TNotifyEvent;`	Вызывается при появлении формы на экране

Класс `TForm` предоставляет разработчику богатый набор средств настройки формы под конкретную задачу. Рассмотрим важнейшие особенности применения свойств и методов класса при выполнении основных операций. Помимо собственных возможностей в форме доступны свойства и методы предков.

Создание и уничтожение формы

В стандартном приложении задача создания и удаления форм возлагается на само приложение. Если разработчик оставил все формы в списке создавае-

мых автоматически *(см. главу 6)*, то после инициализации программы будут выполнены конструкторы всех форм проекта. При этом первой создается главная форма. При окончании работы программы все формы также автоматически уничтожаются, последней идет главная форма проекта.

Самостоятельно создать форму разработчик может с помощью ее конструктора:

```
if not Assigned(Form1)
  then Form1 := TForm1.Create(Application);
```

Удаление созданных таким образом форм осуществляется автоматически.

При необходимости уничтожить форму в еще работающем приложении используется метод `Release`:

```
if Assigned(Form1)
  then Release;
```

Для форм предпочтителен именно этот метод, т. к. он, в отличие от уже знакомого вам метода `Free`, обеспечивает завершение выполнения всех методов-обработчиков, которые могут работать в форме на момент закрытия.

При создании и уничтожении формы соответственно вызываются методы-обработчики `OnCreate` и `OnDestroy`. Обычно они осуществляют инициализацию переменных и элементов управления, создают и удаляют необходимые объекты (листинг 7.2).

Листинг 7.2. Пример работы с формой

```
var SomeList: TStringList;
procedure TForm1.FormCreate(Sender: TObject);
begin
 SomeList := TStringList.Create;
end;
...
{исходный код модуля формы}
...
procedure TForm1.FormDestroy(Sender: TObject);
begin
 try
 finally
  SomeList.Free;
 end;
end;
```

На выполнение рассмотренных методов-обработчиков оказывает влияние свойство OldCreateOrder. При значении False (это значение по умолчанию) метод-обработчик OnCreate выполняется только после всех конструкторов, а метод-обработчик OnDestroy — перед всеми деструкторами.

Для генерации события OnCreate служит метод AfterConstruction. Для OnDestroy — BeforeDestruction.

Визуализация формы

После создания приложение обеспечивает появление на экране только одной главной формы. Все остальные появляются в результате действий пользователя или как реакция на некоторые события в ОС.

Для визуализации формы существуют два метода.

- Show — обеспечивает появление формы и ее работу в немодальном режиме.
- ShowModal — то же, но в модальном режиме.

Закрытие формы обеспечивает метод Close, который сначала вызывает метод CloseQuery, определяющий возможность выполнения операции, если он возвращает True, форма закрывается.

Закрыть форму можно методом-обработчиком OnCloseQuery. В нем на основе действий пользователя или анализа состояния приложения принимается решение о закрытии формы. Для закрытия формы параметр CanClose должен иметь значение True:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
 CanClose :=
   MessageDlg('Закрыть форму?',mtConfirmation,[mbYes,mbNo],0)=mrYes;
end;
```

Для определения состояния видимости формы рекомендуется свойство Visible. Метод Hide изменяет значение свойства Visible на False.

При отображении формы в распоряжении программиста есть метод-обработчик OnShow, который вызывается при использовании методов Show или ShowModal.

Положение формы на экране и ее размеры определяет свойство Position. По умолчанию значение этого свойства равно poDesigned. При этом форма во время выполнения приложения появляется на том же месте и того же размера, что и во время разработки. В обычных приложениях удобно задать значение poScreenCenter, которое определяет размещение формы в центре экрана без изменения размеров.

При переносе приложений на другие компьютеры может возникнуть проблема с масштабированием форм. Дело в том, что на компьютерах могут быть установлены шрифты разных размеров. Если приложение создавалось на основе обычных шрифтов, то при переносе на компьютер, в ОС которого установлены большие шрифты, все тексты на формах приложения увеличатся и будут частично обрезаны.

Свойство `PixelsPerInch` определяет масштаб шрифта. Для стандартных шрифтов его значение равно 96, для больших шрифтов — 120. По значению этого свойства можно определить установленный в системе тип шрифта и провести масштабирование. Для этого свойство `Scaled` должно быть равно `True`.

Изменение значения свойства `PixelsPerInch` во время разработки не имеет смысла, т. к. при компиляции проекта значение автоматически устанавливается в соответствии с установками в системе.

Каждая форма имеет специальное свойство `Canvas`, которое обеспечивает рисование на ее поверхности *(см. главу 12)*. Размеры клиентской (рабочей) области формы, в которой можно размещать компоненты, рисунки и т. д., определяются свойствами `ClientHeight`, `ClientWidth` и `ClientRect`.

При получении сообщения о необходимости перерисовать себя форма вызывает метод-обработчик `OnPaint` (если он определен). Если, например, при помощи свойства `Canvas` нарисовать на форме линию, то при перекрытии формы другими окнами восстановление линии произойдет только при наличии в методе-обработчике `OnPaint` соответствующего кода (листинг 7.3).

Листинг 7.3. Пример использования свойства `Canvas`

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 with Canvas do
 begin
  Pen.Color := clRed;
  Pen.Width := 10;
  MoveTo(10, 10);
  LineTo(ClientWidth – 10, ClientHeight – 10);
 end;
end;
```

При нажатии кнопки по диагонали формы будет проведена красная линия, которую легко стереть, поместив на время поверх формы другое окно. Если

этот же код разместить в методе-обработчике OnPaint, то линия будет отображаться так же надежно, как и любой другой компонент.

Атрибуты и стили формы

Любое окно должно обладать определенным набором стандартных элементов управления. Это кнопка системного меню, кнопки изменения размеров и т. д. Наличие этих элементов на форме определяется свойством BorderIcons.

Значок формы определяется свойством Icon. Он постоянно виден в левой части заголовка формы и отображается при сворачивании формы.

Каждая форма может иметь собственное меню, за которое отвечает свойство Menu.

Кроме этого, существует несколько стилей форм, которые необходимы в различных ситуациях. Стиль формы определяется свойством BorderStyle:

```
type TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog,
bsToolWindow, bsSizeToolWin);

property BorderStyle: TFormBorderStyle;
```

Причем при работе со свойством изменения в форме происходят только при выполнении приложения. Возможные стили форм приведены в табл. 7.2.

Таблица 7.2. *Стили форм*

Тип формы	Описание
bsNone	На форме полностью отсутствуют все системные элементы управления. Форма представляет собой прямоугольник без рамки и заголовка, размеры такой формы нельзя изменить
bsSingle	Стандартное окно без возможности изменения размеров. Набор системных кнопок зависит от свойства BorderIcon
bsSizeable	Значение по умолчанию. Стандартная форма с возможностью изменения размеров. Набор системных кнопок зависит от свойства BorderIcon
bsDialog	Стандартное диалоговое окно. Набор системных кнопок частично зависит от свойства BorderIcon — можно только удалить те кнопки, которые имеются в стандартном диалоге
bsToolWindow	Панель инструментов. Идентична стилю bsSingle, но с уменьшенными системными элементами
bsSizeToolWin	Панель инструментов. Идентична стилю bsSingle, но с уменьшенными системными элементами

Управление компонентами формы

Если форма приложения имеет фокус ввода, то свойство `Active` принимает значение `True`. При этом активный (имеющий фокус ввода) элемент управления определяется свойством

```
property ActiveControl: TWinControl;
```

Если форма не имеет фокуса, то его можно получить при помощи метода `SetFocus`.

Из формы можно получить доступ к любому элементу управления. Для этого имеется свойство `Controls`, унаследованное от класса `TWinControl`:

```
property Controls[Index: Integer]: TControl;
```

От этого же класса унаследованы методы, позволяющие добавлять и удалять компоненты формы программно во время выполнения приложения:

```
procedure InsertControl(AControl: TControl);
procedure RemoveControl(AControl: TControl);
```

Для передачи фокуса компоненту программным способом предусмотрен метод

```
procedure FocusControl(Control: TWinControl);
```

В параметре метода передается указатель на нужный элемент управления.

Кадры (Frames). Класс *TFrame*

Компонент `Frames` находится в категории **Standard** Палитры инструментов.

Что собой представляет кадр? С точки зрения объектно-ориентированного программирования, это класс-потомок `TScrollingWinControl`, т. е. "двоюродный брат" формы. С точки зрения разработчика кадр — это контейнер для других визуальных компонентов, в том числе других кадров, который может находиться в приложении или может быть помещен в Палитру инструментов.

Если вы просто попытаетесь перенести компонент `Frames` на форму, вас ждет разочарование — сообщение "No frames in project. To create a frame select File | New | Frame".

Буквально последовать этому совету невозможно. Дело в том, что это сообщение присутствует в среде разработки, начиная с Delphi 5. За прошедшее время среда разработки много раз изменялась, а это сообщение по недосмотру разработчиков Borland осталось прежним. Поэтому для создания рамки необходимо открыть Репозиторий (команда **File | New | Other**) и в категории **Delphi Projects | Delphi Files** выбрать значок **Frame**. После этого в проекте появится новый модуль и новый кадр, на первый взгляд похожий на форму.

Теперь вы можете наполнять созданный кадр компонентами. Хороший пример применения кадров приводится вместе с Delphi — это приложение FramesDemo в папке ..Demos\DelphiWin32\VCLWin32\Frames. Зачем там кадры? Для доступа и визуализации данных одной таблицы БД требуются компоненты доступа к данным `TTable` и `TDataSource`, компонент отображения данных `TDBGrid` и компонент управления `TDBNavigator`. Если в базе есть поля типа Memo и Graphic, нужно добавить еще `TDBMemo` и `TDBImage`. Теперь, если приложение должно одинаковым образом обеспечивать работу пользователя с несколькими таблицами на одной форме, описанный набор компонентов нужно продублировать нужное число раз. Для этого разработчикам примера и понадобились кадры. От одного базового кадра для работы с данными порождены еще несколько — для решения специфических задач.

Итак, кадр создан. Что дальше? В его всплывающем меню есть пункт **Add to Palette**. При его вызове возникает диалоговое окно (рис.7.2).

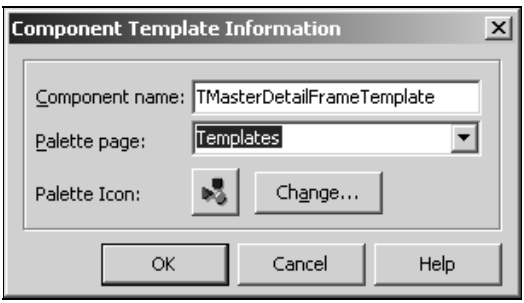


Рис. 7.2. Диалог добавления кадра в Палитру инструментов

Вы можете задать имя нового кадра, его будущую страницу Палитры (по умолчанию это категория **Templates**) и значок. Специфика его в том, что все изменения в кадре будут немедленно отображаться в использующих его проектах. Если вы захотите вызвать кадр, он появится в вашем проекте не в том состоянии, в котором находился в момент помещения в Палитру, а в своем текущем состоянии.

Резюме

Формы являются основой пользовательского интерфейса любого приложения. Класс `TForm`, инкапсулирующий форму, обладает богатыми возможностями по управлению компонентами, размещенными на ней.

Наиболее употребительные формы разработчик может поместить в Репозиторий и затем использовать в других проектах. Это позволяет экономить время на разработку, тиражировать удачные решения и создавать приложения со схожими интерфейсами.

Глава 8

Элементы управления Win32 и XP

Элементы управления составляют суть пользовательского интерфейса приложений Windows. Всеми программами нужно управлять более или менее единообразно, поэтому в составе операционной системы имеется набор типовых кнопок, редактирующих элементов, списков выбора и т. п., которыми вы можете "украсить" свои разработки. Типовые элементы управления содержатся в библиотеке Comctl32.dll.

В Delphi 2005 основные визуальные компоненты для разработки пользовательского интерфейса приложений собраны в трех категориях **Standard**, **Additional** и **Win32** Палитры инструментов. В этой главе рассматриваются только некоторые элементы управления, отобранные из соображений принадлежности к системной динамической библиотеке ComCtl32 и наибольшей сложности использования. Все они расположены в категории **Win32** Палитры инструментов.

Все элементы управления могут отображаться в двух стилях: в традиционном стиле для приложений Windows, подобно пользовательскому интерфейсу операционных систем Windows 2000 или Windows 2003, или в стиле операционной системы Windows XP.

В этой главе рассматриваются следующие вопросы.

- Системная динамическая библиотека ComCtl32.
- Обзор возможностей элементов управления Win32.
- Представление элементов управления в стиле Windows XP.

Что такое библиотека ComCtl32

Когда-то давно, изменив внешний вид "окон" в Windows 95, менеджеры Microsoft задумались о том, чтобы дать независимым разработчикам средства для создания приложений, внешне похожих на системные утилиты и использующих единые типовые элементы управления. Например, все эти элементы автоматически поддерживают установленные в системе цветовое оформление, размер, шрифт и т. п.

Все элементы, проверенные и обкатанные в Windows, объединялись в библиотеке comctl32.dll, документировались и публиковались для разработчиков. С 1995 года сменилось много версий библиотеки, элементы добавлялись и совершенствовались. Соответственно росла и страница Win32 в Палитре инструментов Delphi. Все компоненты, представленные там, взяты из comCtl32.

Специально отметим, что эти элементы управления не являются ActiveX. Это обычные специализированные разновидности окон Windows, установка свойств которых происходит через посылку специализированных сообщений. Полная документация по всем сообщениям и применяемым в них константам и структурам есть в MSDN. Для большинства сообщений предусмотрены специальные оболочечные функции, которые описаны в модуле CommCtrl.pas. Сами классы компонентов Delphi описаны в модуле ComCtrls.pas.

Примечание

Своими свойствами компоненты Delphi покрывают лишь 70—80% возможностей соответствующих элементов управления. Так что для решения некоторых специфических задач иногда приходится обращаться к функциям ComCtrl.pas. Подобные примеры имеются как в этой, так и в последующих главах книги.

Когда вы распространяете свое приложение, содержащее компоненты со страницы **Win32** Палитры компонентов, не забудьте озаботиться проблемой совместимости. Если версия библиотеки comctl32 на компьютере пользователя старее той, что применялась вами при разработке, то в лучшем случае вы получите проблемы с правильной отрисовкой и поведением элементов управления, а в худшем — фатальную ошибку и неработающее приложение.

Обновление библиотеки может быть скачано с сервера компании Microsoft по адресу **http://www.microsoft.com/msdownload/ieplatform/ie/comctrlx86.asp**. Ни в коем случае не пытайтесь просто ("вручную") заменить comctl32.dll на более новую версию — это запрещено Microsoft, и последствия могут вас серьезно огорчить.

Есть и более общий способ решения этой проблемы — установить последнюю версию Internet Explorer, который включает в себя новую comctl32. Если у вас дело упирается только в элементы управления, то этот способ избыточен. Но если в вашем распространяемом приложении есть другие решения от Microsoft, привязанные к IE, то он может оказаться необходимым.

Проверка версии библиотеки элементов управления выполняется с помощью функции из модуля ComCtrls.pas:

```
function GetComCtlVersion: Integer;
```

Она возвращает номер установленной версии библиотеки в виде пары цифр. Каждая версия выходит обычно вместе со следующей версией обозревателя Internet. Вот константы, описанные также в ComCtrls.pas:

```
const
  ComCtlVersionIE3 = $00040046;
  ComCtlVersionIE4 = $00040047;
  ComCtlVersionIE401 = $00040048;
  ComCtlVersionIE5 = $00050050;
  ComCtlVersionIE501 = $00050051;
  ComCtlVersionIE6 = $00060000;
```

В Windows 2000 и ME библиотека элементов управления имеет версию `$00050051`, а в Windows XP сделаны кардинальные изменения внешних возможностей интерфейса пользователя. Их основой является 6-я версия библиотеки comctl32.

И еще немного о совместимости. Не нужно быть мудрецом, чтобы понять, что компоненты со страницы **Win32** работают только в среде Win32. Поэтому если вы создаете приложение для Linux или .NET, забудьте о библиотеке comctl32 и категории Палитры инструментов **Win32** и пропустите эту главу.

Многостраничный блокнот — компоненты *TPageControl* и *TTabControl*

В Палитре инструментов имеется два элемента управления, обеспечивающих создание многостраничных блокнотов. Это компоненты `TTabControl` и `TPageControl`. Переключение между страницами осуществляется при помощи закладок. Закладки могут выглядеть как "настоящие" в бумажном блокноте, а могут быть похожи на стандартные кнопки Windows.

Компонент `TTabControl` отличается тем, что представляет собой как бы "виртуальный" блокнот. Это единый объект с одной фактической страницей. При переключении закладок осуществляется вызов метода-обработчика события

```
property OnChange: TNotifyEvent;
```

соответствующий код в котором может изменить набор видимых элементов управления и создать для пользователя иллюзию "переключения страниц".

Компонент `TPageControl` является контейнером для объектов `TTabSheet`, соответствующих отдельным страницам блокнота. Каждая страница в нем может нести свой набор дочерних компонентов, их можно переключать уже во время разработки.

Первый подход удобен, если на разных страницах у вас должны располагаться одни и те же компоненты, "начиненные" разными данными. При переключении закладок происходит навигация по таблице, содержимое полей меняется и создается впечатление перехода на другую страницу.

Второй подход необходим, если у вас действительно разные страницы с разными наборами компонентов на них. `TPageControl` используют для создания редакторов свойств и настроек программы, а также для разного рода мастеров (Wizards).

Оба компонента в своей основе имеют общий элемент управления из comctl32 (в документации Microsoft он называется `Tab Control`). Соответственно, в иерархии классов Delphi они оба произошли от класса `TCustomTabControl`, от которого унаследовали значительную часть свойств и методов. А вот механизмы работы отдельных страниц у каждого компонента свои. Поэтому сначала мы рассмотрим общие для двух компонентов свойства, а затем особенности работы со страницами. Свойства и методы-обработчики класса-предка `TCustomTabControl` представлены в табл. 8.1. Обратите внимание, что перечисленные свойства и методы в потомках объявляются как опубликованные (`Published`).

Таблица 8.1. *Основные свойства и методы-обработчики, общие для* `TTabControl` *и* `TPageControl`

Объявление	Описание
`property TabIndex: Integer;`	Задает номер текущей страницы, начиная с 0
`property TabHeight: Smallint;`	Задает высоту закладок в пикселах. При значении 0 высота определяется автоматически так, чтобы вместить текст
`property TabWidth: Smallint;`	Задает ширину закладок. При значении 0 ширина определяется автоматически так, чтобы вместить текст
`type TTabStyle = (tsTabs, tsButtons, tsFlatButtons);` `property Style: TTabStyle;`	Определяет стиль закладок компонента: `tsTabs` — стандартные закладки; `tsButtons` — объемные кнопки; `tsFlatButtons` — плоские кнопки
`type TTabPosition = (tpTop, tpBottom, tpLeft, tpRight);` `property TabPosition: TTabPosition;`	Определяет расположение закладок на компоненте. Расположение, отличное от `tpTop`, возможно только для стиля `tsTabs`

Таблица 8.1 *(окончание)*

Объявление	Описание
`property HotTrack: Boolean;`	При значении `True` названия страниц выделяются цветом при перемещении над ними указателя мыши
`property Images: TCustomImageList;`	Указывает на список картинок, появляющихся на закладках страниц
`property RaggedRight: Boolean;`	При значении `True` ширина закладок изменяется таким образом, чтобы они не занимали всю сторону блокнота
`property MultiLine: Boolean;`	При значении `True` закладки страниц могут располагаться в несколько рядов (если они не помещаются в один). При значении `False` в верхнем правом углу появляются кнопки, организующие прокрутку невидимых заголовков
`property ScrollOpposite: Boolean;`	При значении `True`, если закладки расположены в несколько рядов, при переходе к закладке следующего ряда все остальные ряды перемещаются на противоположную сторону блокнота. Действительно только при `MultiLine=True`
`type TTabChangingEvent = procedure (Sender: TObject; var AllowChange: Boolean) of object;` `property OnChanging: TTabChangingEvent;`	Вызывается непосредственно перед открытием новой страницы. Параметр `AllowChange`, установленный в `False`, запрещает открытие
`property OnChange: TNotifyEvent;`	Вызывается при открытии новой страницы
`property OnDrawTab: TDrawTabEvent;`	Вызывается при перерисовке страницы, только если свойство `OwnerDraw` = `True`
`property OnGetImageIndex: TtabGetImageEvent;`	Вызывается при отображении на закладке картинки

Как видно из табл. 8.1, большинство свойств обеспечивают различные стили представления многостраничного блокнота. При настройке стиля обратите внимание, что свойство `RaggedRight` может не работать, т. к. вступает в противоречие со свойством `TabWidth`. При `TabWidth = 0` компонент изменяет ширину закладок в соответствии с длиной текста, в противном случае ширина закладок всегда равна значению свойства `TabWidth`.

Для того чтобы в закладках совместно с текстом показать картинки, предусмотрено свойство `Images`, в котором необходимо задать требуемый экземпляр компонента `TImageList`.

Свойство `TabIndex`, задающее номер текущей страницы, позволяет переключать страницы программно. Для компонента `TTabControl` это единственный способ изменить текущую страницу на этапе разработки. При смене страниц сначала происходит событие `OnChanging` — в этот момент `TabIndex` еще содержит индекс старой страницы (и смену можно запретить), а затем `OnChange` — это свойство уже указывает на новую страницу.

В компоненте `TTabControl` число и заголовки страниц полностью зависят от свойства

```
property Tabs: TStrings;
```

В списке перечисляются заголовки страниц, для которых автоматически создаются закладки. Порядок следования страниц зависит от расположения текстов заголовков в свойстве `Tabs`.

При этом забота о правильном чередовании элементов управления при смене страниц полностью ложится на программиста. Для этого необходимо в методе-обработчике `OnChange` определить видимость элементов в зависимости от индекса текущей страницы:

```
procedure TForm1.TabControl1Change(Sender: TObject);
begin
 with TabControl1 do
 begin
  Edit1.Visible := TabIndex = 0;
  Edit2.Visible := TabIndex = 1;
  Edit3.Visible := TabIndex = 2;
 end;
end;
```

Компонент `TPageControl`, в отличие от `TTabControl`, для обеспечения работы создает "настоящую" страницу — экземпляр класса `TTabSheet`. Список указателей на все созданные экземпляры страниц хранится в свойстве `Pages`, доступном только для чтения:

```
property Pages[Index: Integer]: TTabSheet;
```

Номер индекса соответствует порядковому номеру страницы. Новая страница создается командой **New Page** из всплывающего меню компонента, перенесенного на форму. Если же вы хотите создать страницу на этапе выполнения,

создайте экземпляр `TTabSheet` самостоятельно и в свойстве `PageControl` укажите на родительский блокнот:

```
 pcMain: TPageControl;
 ts : TTabSheet;
...
ts := TTabSheet.Create( pcMain );
  with ts do
   begin
    PageControl := pcMain;
    ts.Caption :='New page';
   end;
```

Общее число страниц хранится в свойстве

```
property PageCount: Integer;
```

доступном только для чтения. Текущую страницу можно задать свойством:

```
property ActivePage: TTabSheet;
```

Если во время разработки (этой возможностью компонент `TPageControl` отличается от своего собрата) или во время выполнения переключиться на другую страницу, значение свойства `ActivePage` изменится.

Для перехода на соседнюю страницу программными средствами можно также использовать метод

```
procedure SelectNextPage(GoForward: Boolean);
```

в котором параметр `GoForward` при значении `True` задает переход на следующую страницу, иначе — на предыдущую.

Рассмотрев свойства блокнота, обратимся к его страницам и остановимся подробнее на возможностях класса `TTabSheet`. На владельца страницы указывает значение свойства

```
property PageControl: TPageControl;
```

Расположение страницы в блокноте задает свойство `PageIndex`:

```
property PageIndex: Integer;
```

Если в блокноте одновременно выделено несколько страниц, то положение данной страницы среди выделенных определяется свойством только для чтения

```
property TabIndex: Integer;
```

Страница может временно "исчезнуть" из блокнота, а затем опять появиться. Для этого применяется свойство

```
property TabVisible Boolean;
```

Компонент *TToolBar*

Возможность создать панель инструментов появилась у разработчика давно, начиная с первых версий Delphi. Тогда она была реализована с помощью сочетания компонентов `TPanel` и `TSpeedButton`. Так можно было поступить и сейчас, но панель инструментов получила развитие с появлением стандартного элемента управления `TToolBar`, который объединяет расположенные на нем кнопки и другие элементы управления и централизованно управляет ими.

Все кнопки (класс `TToolButton`) на панели инструментов имеют одинаковый размер, задаваемый свойствами:

- `property ButtonWidth: Integer;`
- `property ButtonHeight: Integer;`

Но эти свойства срабатывают только тогда, когда свойство

```
property ShowCaptions: Boolean;
```

имеет значение `False`. Оно отвечает за видимость надписей на кнопках компонента. И если эти надписи должны быть видимы, то размер всех кнопок автоматически подстраивается под размер кнопки с самым длинным текстом.

На каждой кнопке могут отображаться два ее атрибута: текст (заголовок кнопки, свойство `Caption`) и картинка. Показ текста можно запретить установкой свойства `ShowCaptions` в `False`.

Панель инструментов тяжело себе представить без украшающих ее картинок. У `TToolBar` целых три свойства, ссылающихся на списки картинок:

- `property Images: TCustomImageList;`
- `property DisabledImages: TCustomImageList;`
- `property HotImages: TCustomImageList;`

В обычном состоянии на кнопках отображаются картинки из набора, указанного свойством `Images`. Если кнопка неактивна (свойство `Enabled` обращено в `False`), надпись на кнопке отображается серым цветом и на ней показывается картинка из свойства `DisabledImages`.

Если свойство

```
property Flat: Boolean;
```

установлено в `True`, внешний вид панели инструментов становится более "модным" — плоским. В этом случае границы кнопок не видны, и все они выглядят как набор надписей и рисунков на единой плоской панели. Границы становятся видны, только когда над кнопкой находится указатель мыши. При этом можно изменить и внешний вид кнопки. Если задано зна-

чение свойства HotImages, то на текущей кнопке обычная картинка из Images меняется на картинку из HotImages. Посмотрите, например, на панель Microsoft Internet Explorer, там все картинки на кнопках серые, но кнопка, к которой подведен указатель мыши, становится цветной.

Текст и картинка на кнопке могут располагаться друг относительно друга двумя способами, в зависимости от значения свойства List. Если List равно False (установка по умолчанию), то картинка располагается сверху, текст — снизу. В противном случае вы увидите текст справа от картинки.

Панели инструментов — это контейнеры, но и они часто располагаются в контейнерах — компонентах TCoolBar или TControlBar. Те, как правило, имеют свою фоновую картинку, и располагающийся сверху компонент TToolBar можно сделать прозрачным. За это отвечает свойство:

```
property Transparent: Boolean;
```

Оно особенно удобно, когда установлен режим плоской панели. В этом случае прозрачен не только фон самой панели, но и всех кнопок на ней.

Перейдем к рассмотрению функциональных возможностей кнопок. Вы думаете, "функциональные возможности" — это громко сказано? С одной стороны, да. Кнопка — это всего лишь то, на что нажимает пользователь, и не более. Главное событие и для кнопки TToolButton, и для панели TToolBar — событие OnClick. Кроме него они могут отреагировать только на перемещение мыши и на процессы перетаскивания/переноса (Drag-and-drop, drag-and-dock; описанные далее в этой главе).

С другой стороны, кнопки можно нажимать в разнообразных вариантах и сочетаниях. Ключ к выбору варианта — свойство Style объекта TToolButton:

```
type TToolButtonStyle = (tbsButton, tbsCheck, tbsDropDown, tbsSeparator,
tbsDivider);
property Style: TToolButtonStyle;
```

Стили tbsSeparator и tbsDivider предназначены для оформления панели и представляют собой пустое место и вертикальный разделитель соответственно. Обычная кнопка — это, понятное дело, стиль tbsButton.

Если вы хотите создать одну или несколько кнопок, "залипающих" после нажатия, выберите стиль tbsCheck. После щелчка такая кнопка остается в нажатом состоянии до следующего нажатия. Об ее состоянии говорит свойство:

```
property Down: Boolean;
```

Если нужна группа кнопок, из которых только одна может пребывать в нажатом состоянии, следует воспользоваться свойством:

```
property Grouped: Boolean;
```

Такая группа называется группой с зависимым нажатием. Если на панели инструментов есть ряд расположенных подряд кнопок с `Style=tbsCheck` и `Grouped=True`, то он будет обладать свойствами группы с зависимым нажатием. Если групп зависимых кнопок должно быть две и более, разделить их между собой можно кнопкой другого стиля (например, `tbsSeparator` или `tbsDivider`) или любым другим элементом управления.

В такой группе всегда должна быть нажата хотя бы одна кнопка, на этапе разработки установите ее свойство `Down` в `True`. Но если это вам не подходит, можно установить свойство

```
property AllowAllUp: Boolean;
```

в `True` и отжимать все кнопки. Значение этого свойства всегда одинаково для всех кнопок в группе.

Если в какой-то ситуации одна или несколько кнопок должны стать недоступными, для этого можно установить свойство `Enabled` в `False`. Но у кнопок в группе есть еще и третье состояние — неопределенное:

```
property Indeterminate: Boolean;
```

Такие кнопки выделяются серым цветом, чтобы показать пользователю, что их выбирать не следует. Переход в состояние `Indeterminate=True` все еще позволяет кнопке обрабатывать событие `OnClick`, но при этом она переходит в отжатое состояние (`Down=False`). Правда, только до следующего нажатия. После него кнопка выходит из состояния `Indeterminate`.

Свойство

```
property Marked: Boolean;
```

отображает поверхность кнопки синим цветом (точнее, цветом `clHighlight`), как у выделенных объектов. В отличие от предыдущего случая с `Indeterminate`, кнопка остается в состоянии `Marked` независимо от нажатий, вплоть до присвоения этому свойству значения `False`.

Далее приведен фрагмент программы (листинг 8.1), с помощью которого можно выделить кнопки на панели при помощи мыши. Приведенные обработчики событий нужно присвоить всем кнопкам панели и самой панели `TToolBar`.

Листинг 8.1. Пример использования кнопок

```
var StartingPoint : TPoint;
    Selecting : boolean;
procedure TForm1.ToolBar1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
 StartingPoint := (Sender as TControl).ClientToScreen(Point(X,Y));
 Selecting := True;
end;
procedure TForm1.ToolBar1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var i: Integer;r,r0 : TRect;
begin
 if Selecting then
  begin
  r.TopLeft := StartingPoint;
  r.BottomRight := (Sender as TControl).ClientToScreen(Point(X,Y));
  with ToolBar1 do for i := 0 to ButtonCount-1 do
   begin
   r0 :=Buttons[i].ClientRect;
   OffsetRect(r0,Buttons[i].ClientOrigin.X,Buttons[i].ClientOrigin.Y);
   if IntersectRect(r0,r,r0) then
     Buttons[i].Marked := True;
   end;
  end;
 Selecting := False;
end;
```

Наличие обработчиков событий `OnMouseDown/OnMouseUp` не мешает нажатию кнопок, оно все равно вызывает событие `OnClick`.

Компонент `TToolBar` может стать полноценной заменой главного меню (взгляните хотя бы на приложения из состава MS Office). К каждой из кнопок можно присоединить меню — и не одно, а целых два:

- `property DropdownMenu: TPopupMenu;`
- `property PopupMenu: TPopupMenu;`

Для того чтобы по нажатию левой кнопки мыши выпадало меню `DropdownMenu`, нужно установить один из стилей кнопок: `tbsButton` или `tbsDropdown`. В первом случае меню возникнет при нажатии в любой части кнопки. При этом событие `OnClick` не возникает; кнопка из-за этого становится "неполноценной" — она пригодна только для показа меню. Второй случай — стиль `tbsDropdown` — специально предназначен для удобства работы с выпадающими меню. Меню возникает при щелчке на специальной области с изображением треугольника в правой части кнопки. А вот щелчок на остальной части кнопки, как обычно, вызовет событие `OnClick`.

Компонент *TImageList*

С ростом возможностей пользовательского интерфейса Windows все больше и больше элементов управления стали оснащаться значками и картинками. И вот для централизованного управления этими картинками появился элемент управления `TImageList`. Он представляет собой оболочку для реализации коллекции одинаковых по размеру и свойствам изображений. На этапе разработки ее "начиняют" картинками. Компонент `TImageList` обладает двумя свойствами: `Images` и `ImageIndex`. Первое указывает на список (компонент `TImageList`), второе — на конкретную картинку в этом списке.

Проще всего заполнить список при помощи встроенного редактора (рис. 8.1), выполнив двойной щелчок на компоненте или выбрав пункт **Image List Editor** в его контекстном меню.

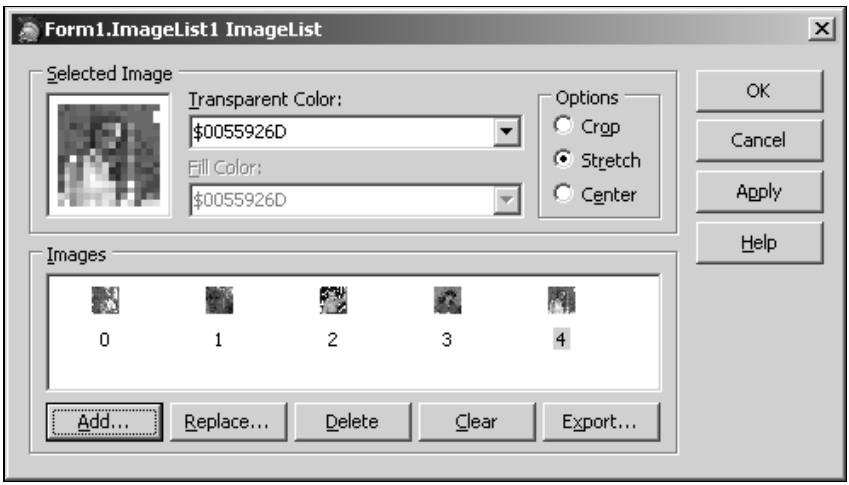


Рис. 8.1. Редактор списка изображений `TImageList`

Пользоваться им очень просто, но нужно обратить внимание на одну тонкость. Только что выбранное изображение можно отредактировать, изменив его положение относительно отведенного ему прямоугольника: **Crop** (размещение начиная с точки (0, 0)), **Stretch** (масштабирование) или **Center** (центровка). Кроме того, можно применить прозрачный цвет (**Transparent Color**). Точки с этим цветом при отрисовке не будут видны (прозрачны). Его можно выбрать либо из списка, либо мышью, щелкнув в нужном месте на увеличенной картинке в верхнем левом углу редактора. Если редактор уже записал изображение в список, редактирование этих свойств становится невозможным. Запись происходит, например, при закрытии редактора. Кроме того, размер картинок (свойства `Height` и `Width`) нужно установить заранее.

Если компонент настроен на размер 16×16, а вы пытаетесь наполнить его картинками 32×32, они будут сжаты и потеряют в внешнем виде.

Можно сильно упростить подбор картинок для `TImageList`. Если просмотреть ресурсы приложений из состава MS Office, да и многих других пакетов, то можно обнаружить, что картинки, которые встречаются на панелях инструментов, "склеены" между собой (рис. 8.2).

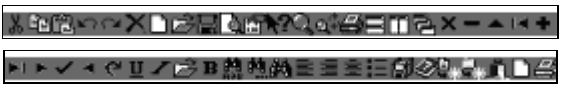


Рис. 8.2. Такие картинки часто используют для панелей инструментов

Их удобно использовать и в собственных программах. Последовательность действий для этого следующая.

1. Создать исходный файл ресурсов, куда нужно включить и поименовать требуемые файлы с расширением bmp, к примеру:

   ```
   inout BITMAP "inout.bmp"
   tools BITMAP "tools.bmp"
   ```

   Сохранить этот файл с расширением rc, скажем, bitmap.rc.

2. Скомпилировать ресурсы при помощи утилиты brcc32.exe, поставляемой с Delphi и размещенной в папке ..\bin.

3. Появившийся файл bitmap.res нужно включить в состав проекта. Для этого служит директива `$R`:

   ```
   {$R bitmap.res}
   ```

4. Теперь картинка содержится в ресурсах и будет включена в состав исполняемого файла. Осталось загрузить ее в компонент `TImageList`. Для этого есть метод `ResourceLoad`:

   ```
   ImageList1.ResourceLoad(rtBitmap, 'bitmaps',TColor(0));
   ```

При этом произойдет автоматическая "нарезка" картинок в соответствии со свойствами `Width` и `Height`. Если размер большой картинки, к примеру, 256×16 пикселов, а ширина, заданная свойством `TImageList`, равна 16 пикселов, то в список будут включены 16 элементов размером 16×16. Поэтому еще во время разработки нужно правильно настроить размеры в компоненте `TImageList`, иначе после загрузки ресурса картинки будут разрезаны как попало.

Есть и другой метод загрузки — `FileLoad`:

```
function FileLoad(ResType: TResType; Name: string; MaskColor: TColor):
Boolean;
```

Аналогичным путем он позволяет загружать картинки из любого пригодного файла. Но загрузка из файла менее надежна, поскольку нет гарантии, что у пользователя вашего приложения нужный файл всегда находится на месте и он не изменен.

Описанный редактор списка картинок "умеет" делать их прозрачными еще во время разработки. Часто бывает необходимо сделать прозрачными картинки, загружаемые из файлов во время исполнения. Для этого нужно применить их свойство `Transparent`:

```
Var bmp: TBitmap;
...
bmp.LoadFromFile('c:\test.bmp');
bmp.Transparent := True;
ImageList1.AddMasked(bmp, bmp.TransparentColor);
...
```

В методе `AddMasked` вторым параметром необходимо указать "прозрачный" (фоновый) цвет, который в данном случае равен `bmp.TransparentColor`.

Как элемент управления Win32, `TImageList` имеет собственный дескриптор:

```
property Handle: HImageList;
```

Не следует путать этот дескриптор с дескрипторами растровых картинок, входящих в состав списка. В файле COMMCTRL.PAS приведены прототипы всех функций для работы с этим элементом управления, и для их вызова необходимо значение свойства `Handle`. Обратитесь к ним, если опубликованных свойств `TImageList` вам недостаточно.

Компоненты *TTreeView* и *TListView*

Эти компоненты известны каждому, кто хоть раз видел Windows. Именно на их базе создано ядро пользовательского интерфейса — оболочка Explorer, да и большинство других утилит Windows. Они включены в COMCTL32.DLL и доступны программистам.

Компонент `TTreeView` называют деревом (рис. 8.3).

Он предназначен для отображения иерархической информации. Его "сердцем" является свойство

```
property Items: TTreeNodes;
```

Это список всех вершин дерева, причем список, обладающий дополнительными полезными свойствами. Каждый из элементов списка — это объект типа `TTreeNode`. Свойства его сведены в табл. 8.2.

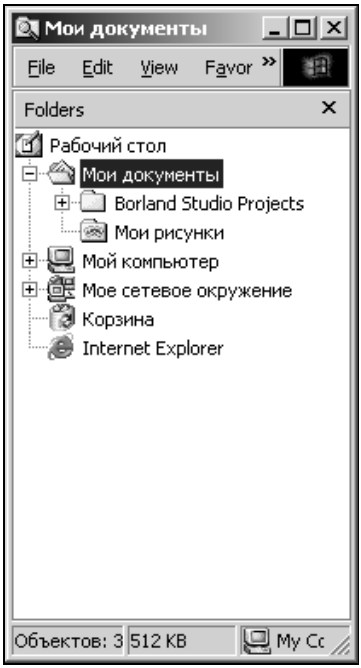


Рис. 8.3. Внешний вид компонента `TTreeView`

Таблица 8.2. *Список свойств объекта* `TTreeNode`

Объявление	Описание
`property HasChildren: Boolean;`	Равно `True`, если узел имеет дочерние узлы
`property Count: Integer;`	Счетчик числа дочерних узлов данного узла
`property Item[Index: Integer]: TTreeNode;`	Список дочерних узлов
`property Parent: TTreeNode;`	Ссылка на объект — родительский узел (верхнего уровня)
`property Level: Integer;`	Уровень, на котором находится узел. Для корневого узла это свойство равно 0; его потомки имеют значение `Level=1` и т. д.
`property Text: string;`	Текст узла
`property Data: Pointer;`	Данные, связанные с узлом
`property TreeView: TCustomTreeView;`	Ссылка на компонент `TTreeView`, в котором отображается данный узел

Таблица 8.2 *(окончание)*

Объявление	Описание
`property Handle: HWND;`	Дескриптор окна компонента `TTreeView`, в котором отображается данный узел
`property Owner: TTreeNodes;`	Ссылка на компонент `TTreeNodes`, которому принадлежит данный узел
`property Index: Longint;`	Индекс узла в списке своего родителя
`property IsVisible: Boolean;`	Равно `True`, если узел видим (все его родительские узлы развернуты)
`property ItemId: HTreeItem;`	Дескриптор узла (применяется при вызове некоторых методов)
`property AbsoluteIndex: Integer;`	Абсолютный индекс узла в списке корневого узла
`property ImageIndex: Integer;`	Индекс картинки, соответствующей невыбранному узлу в нормальном состоянии
`property SelectedIndex: Integer;`	Индекс картинки, соответствующей выбранному узлу
`property OverlayIndex: Integer;`	Индекс картинки, которая может накладываться поверх основной
`property StateIndex: Integer;`	Индекс дополнительной картинки, отражающей состояние узла
`property Selected: Boolean;`	Равно `True`, если данный узел выбран пользователем
`property Focused: Boolean;`	Равно `True`, если данный узел выбран пользователем для редактирования текста узла
`property Expanded: Boolean;`	Равно `True`, если данный узел развернут (показываются его дочерние узлы)

Очень важным является свойство `Data`. Вместе с каждым узлом можно хранить не только текст, но и любые данные. Следует только помнить, что при удалении узла они автоматически не освобождаются, и это придется сделать вручную.

Для добавления узлов в дерево существуют десять методов объекта `TTreeNode` (табл. 8.3).

Таблица 8.3. *Методы, позволяющие добавлять узлы в объект* `TTreeNode`

№	Метод	Смысл
1	`function Add(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется последним в тот же список, что и узел `Node`
2	`function AddObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и п. 1, но с узлом связываются данные из параметра `Ptr`
3	`function AddFirst(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется первым в тот же список, что и узел `Node`
4	`function AddObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и п. 3, но с узлом связываются данные из параметра `Ptr`
5	`function AddChild(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется последним в список дочерних узлов узла `Node`
6	`function AddChildObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и п. 5, но с узлом связываются данные из параметра `Ptr`
7	`function AddChildFirst(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется первым в список дочерних узлов узла `Node`
8	`function AddChildObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и п. 7, но с узлом связываются данные из параметра `Ptr`
9	`function Insert(Node: TTreeNode; const S: string): TTreeNode;`	Узел добавляется непосредственно перед узлом `Node`
10	`function InsertObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	То же, что и п. 9, но с узлом связываются данные из параметра `Ptr`

Во всех этих методах параметр `S` — это текст создаваемого узла. Место появления узла (первый или последний) также зависит от состояния свойства `TTreeView.SortType`:

```
type TSortType = (stNone, stData, stText, stBoth);
property SortType: TSortType;
```

Если узлы дерева как-либо сортируются, то новые появляются сразу в соответствии с правилом сортировки. По умолчанию значение этого свойства равно `stNone`.

Добавляя к дереву сразу несколько узлов, следует воспользоваться парой методов `BeginUpdate` и `EndUpdate`:

```
TreeView1.Items.BeginUpdate;
ShowSubKeys(Root,1);
TreeView1.Items.EndUpdate;
```

Они позволяют отключать и снова включать перерисовку дерева на экране на момент добавления (удаления, перемещения) узлов и тем самым сэкономить подчас очень много времени.

Помимо добавления узлов в дерево программным способом, можно сделать это и вручную, во время разработки. При щелчке в Инспекторе объектов на свойстве `Items` запускается специальный редактор (рис. 8.4).

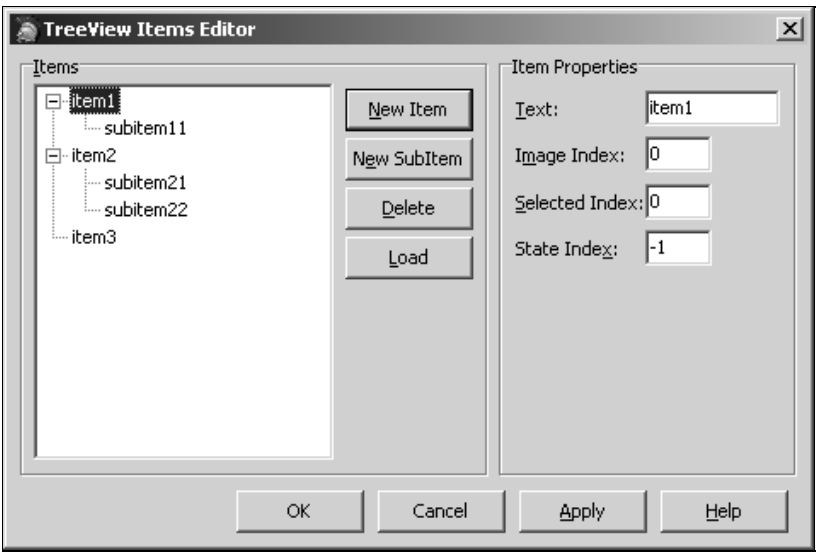


Рис. 8.4. Внешний вид редактора узлов компонента `TTreeView`

Внешний вид компонента `TTreeView` может быть весьма основательно настроен под нужды пользователя. Свойство `ShowButtons` отвечает за то, будут ли показываться кнопки со значками "+" и "–" перед узлами, имеющими "потомство" (дочерние узлы). Щелчок на этих кнопках позволяет сворачивать/разворачивать дерево дочерних узлов. В противном случае делать это нужно двойным щелчком на узле или установить свойство `AutoExpand` в `True` — тогда сворачивание и разворачивание будет происходить автоматически при выделении узлов. Свойство `ShowLines` определяет, будут ли родительские и дочерние узлы соединяться видимыми линиями. Аналогично,

ShowRoot определяет, будут ли на рисунке соединяться между собой линиями корневые узлы (если их несколько). При помощи свойства HotTrack можно динамически отслеживать положение текущего узла: если оно установлено в True, то текущий узел (не выделенный, а именно текущий — тот, над которым находится курсор мыши) подчеркивается синей линией.

Наконец, для оформления дерева можно использовать наборы картинок в свойствах Images и StateImages. Напомним, что у каждого узла-объекта TTreeNode есть свойства ImageIndex и StateIndex, а вдобавок еще и SelectedIndex. Первая картинка предназначена для отображения типа узла, а вторая — его состояния. Можно сразу (при добавлении) указать номера изображений для того или иного случая, а можно делать это динамически. Для этого существуют события:

```
type TTVExpandedEvent = procedure(Sender: TObject; Node: TTreeNode) of
object;
property OnGetImageIndex: TTVExpandedEvent;
property OnGetSelectedIndex: TTVExpandedEvent;
```

Пример их дан в листинге 8.2: в момент возникновения этих событий следует переопределить свойство ImageIndex или SelectedIndex передаваемого в обработчик события объекта TTreeNode.

Свойства StateIndex и StateImages можно порекомендовать для имитации множественного выбора. Дело в том, что, в отличие от TListView, в TTreeView невозможно одновременно выбрать несколько узлов. Вы можете отслеживать щелчки на узлах, и для выбранных устанавливать значение StateIndex в 1; под этим номером в StateImages поместите, например, галочку.

Изложенная информация будет неполной, если не рассказать, на какие события реагирует компонент TTreeView. Большинство из них происходит парами: до наступления какого-то изменения и после него. К примеру, возникновение события OnChanging означает начало перехода фокуса от одного узла к другому, а OnChange — его завершение.

Четверка событий

```
type TTVCollapsingEvent = procedure(Sender: TObject; Node: TTreeNode; var
AllowCollapse: Boolean) of object;
type TTVExpandingEvent = procedure(Sender: TObject; Node: TTreeNode; var
AllowExpansion: Boolean) of object;
property OnExpanding: TTVExpandingEvent;
property OnExpanded: TTVExpandedEvent;
property OnCollapsing: TTVCollapsingEvent;
property OnCollapsed: TTVExpandedEvent;
```

сопровождает процесс свертывания/развертывания узла, а пара

```
type TTVEditingEvent = procedure(Sender: TObject; Node: TTreeNode; var
AllowEdit: Boolean) of object;
property OnEditing: TTVEditingEvent;
type TTVEditedEvent = procedure(Sender: TObject; Node: TTreeNode; var S:
string) of object;
property OnEdited: TTVEditedEvent;
```

сопровождает редактирование его текста. Событие `OnDeletion` происходит при удалении узла. Иногда нужно сравнивать узлы между собой — если вы хотите сделать это по своим правилам, выберите событие `OnCompare`.

Наконец, те, кому и приведенных возможностей мало, могут сами рисовать на компоненте `TTreeView`. У него есть свойство `Canvas` и четыре события — `OnCustomDraw`, `OnCustomDrawItem`, `OnAdvancedCustomDraw`, `OnAdvancedCustomDrawItem`.

Перейдем теперь к компоненту `TListView`. Его еще называют расширенным списком. Действительно, в этот компонент заложено столько, что он перекрывает все мыслимые и немыслимые задачи по отображению упорядоченной однородной информации.

Начнем со свойства `ViewStyle`:

```
type TViewStyle = (vsIcon, vsSmallIcon, vsList, vsReport);
property ViewStyle: TViewStyle;
```

В зависимости от значения этого свойства кардинально меняется внешний вид компонента. Описание значений приведено в табл. 8.4.

Таблица 8.4. *Режимы отображения компонента* `TListView`

Значение	Внешний вид
`vsIcon`	Элементы списка появляются в виде больших значков с надписью под ними. Картинки для больших значков хранятся в свойстве `LargeImages`. Возможно их перетаскивание
`VsSmallIcon`	Элементы списка появляются в виде маленьких значков с надписью справа. Картинки для маленьких значков хранятся в свойстве `SmallImages`. Возможно их перетаскивание
`vsList`	Элементы списка появляются в колонке один под другим с надписью справа. Перетаскивание невозможно
`vsReport`	Элементы списка появляются в нескольких колонках один под другим. В первой содержится маленький значок и надпись, в остальных — определенная программистом информация. Если свойство `ShowColumnHeaders` установлено в `True`, колонки снабжаются заголовками

Как и для предыдущего компонента, элементы списка содержатся в свойстве `Items`. Это и есть собственно список, ничего необычного, кроме методов добавления/удаления, там нет. Каждый элемент списка, объект `TListItem`, в свою очередь, похож на `TTreeNode`. Но у него есть и важное отличие: он может стать носителем большого количества дополнительной информации. Помимо свойства `Data`, у него есть и свойство

```
property SubItems: TStrings;
```

При помощи этого свойства с каждым элементом списка может быть связан целый набор строк и объектов. Но как эти строки показать пользователю? Именно они должны, по замыслу разработчиков этого элемента управления, отображаться в режиме отображения `vsReport`. Сначала следует создать необходимое количество заголовков колонок (заполнив свойство `Columns`), учитывая, что первая из них будет отведена под сам текст элемента списка (свойство `Caption`). Последующие же колонки будут отображать текст строк из свойства `Items.SubItems` (рис. 8.5).

Columns[0]	Columns[1]	...	Columns[m+1]
Items[0].Caption	Items[0].SubItems[0]	...	Items[0].SubItems[m]
Items[1].Caption	Items[1].SubItems[0]	...	Items[1].SubItems[m]
...	...	...	...
Items[n].Caption	Items[n].SubItems[0]	...	Items[n].SubItems[m]

Рис. 8.5. Так будет располагаться информация компонента `TListView` в режиме `vsReport`

Элементы в списке могут быть отсортированы — за это отвечает свойство `SortType`. Можно отсортировать элементы не только по названию (это возможно при значении `SortType` равном `stText`), но и по данным (значения `stData` и `stBoth`), как это сделано в утилите Explorer. Для реализации такой сортировки нужно обработать события `OnColumnClick` и `OnCompare`:

```
var ColNum : Integer;
procedure TMainForm.ListView1ColumnClick(Sender: TObject; Column:
TListColumn);
begin
 ColNum := Column.Index;
 ListView1.AlphaSort;
end;
procedure TMainForm.ListView1Compare(Sender: TObject; Item1, Item2:
TListItem; Data: Integer; var Compare: Integer);
```

```
begin
if ColNum = 0 then //заголовок
 Compare := CompareStr(Item1.Caption, Item2.Caption);
else
 Compare := CompareStr(Item1.SubItems[ColNum-1],
Item2.SubItems[ColNum-1]);
end;
```

Рассмотрим пример работы с компонентами `TTreeView` и `TListView`. Где их совместное использование будет полезным? Выберем для этого отображение данных системного реестра. С одной стороны, ключи в реестре упорядочены иерархически. С другой, каждый из них может иметь несколько разнотипных значений. Таким образом, мы почти пришли к тому же решению, что и разработчики из Microsoft, создавшие утилиту Registry Editor: слева — дерево ключей, справа — расширенный список их содержимого.

Конечно, нет смысла дублировать их труд. Здесь мы ограничимся только просмотром реестра, затратив на это совсем немного усилий: четыре компонента и пару десятков строк кода. На рис. 8.6 показано, как выглядит главная (и единственная) форма приложения Mini Registry Browser, а его исходный код приведен в листинге 8.2.

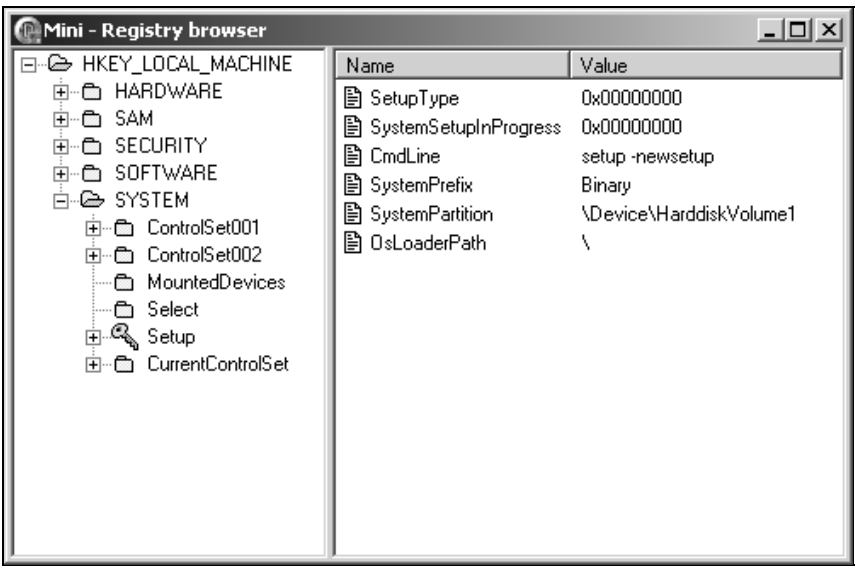


Рис. 8.6. Внешний вид приложения Mini Registry Browser

Листинг 8.2. Приложение Mini Registry Browser, главный модуль

```
unit main;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
logs,
  StdCtrls, Grids, Outline, ComCtrls, ImgList, ExtCtrls;
type
  TForm1 = class(TForm)
    TreeView1: TTreeView;
    ListView1: TListView;
    ImageList1: TImageList;
    Splitter1: TSplitter;
    procedure FormCreate(Sender: TObject);
    procedure TreeView1Change(Sender: TObject; Node: TTreeNode);
    procedure FormDestroy(Sender: TObject);
    procedure TreeView1Expanded(Sender: TObject; Node: TTreeNode);
    procedure TreeView1GetImageIndex(Sender: TObject; Node: TTreeNode);
  private
    { Private declarations }
  public
    { Public declarations }
   procedure ShowSubKeys(ParentNode: TTreeNode;depth: Integer);
   function GetFullNodeName(Node: TTreeNode):string;
  end;
var
  Form1: TForm1;
implementation
uses registry;
{$R *.DFM}
var  reg : TRegistry;
procedure TForm1.FormCreate(Sender: TObject);
var root : TTreeNode;
begin
Reg := TRegistry.Create;
ListView1.ViewStyle := vsReport;
with ListView1 do
```

```
begin
 with Columns.Add do
  begin
   Width := ListView1.Width div 3 - 2;
   Caption := 'Name';
  end;
 with Columns.Add do
  begin
   Width := ListView1.Width div 3 * 2 - 2;
   Caption := 'Value';
  end;
end;
 TreeView1.Items.Clear;
 Reg.RootKey := HKEY_LOCAL_MACHINE;
 Root := TreeView1.Items.Add(nil,'HKEY_LOCAL_MACHINE');
 TreeView1.Items.AddChild(root,'');
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
 Reg.Free;
end;
function TForm1.GetFullNodeName(Node: TTreeNode):string;
var CurNode : TTreeNode;
begin
 Result:=''; CurNode := Node;
 while CurNode.Parent<>nil do
 begin
  Result:= '\'+CurNode.Text + Result;
  CurNode := CurNode.Parent;
 end;
end;
procedure TForm1.TreeView1Change(Sender: TObject; Node: TTreeNode);
var s: string;
KeyInfo : TRegKeyInfo;
ValueNames : TStringList;
i : Integer;
DataType : TRegDataType;
begin
 ListView1.Items.Clear;
```

```
 s:= GetFullNodeName(Node);
 if not Reg.OpenKeyReadOnly(s) then Exit;
 Reg.GetKeyInfo(KeyInfo);
 if KeyInfo.NumValues<=0 then Exit;
 ValueNames := TStringList.Create;
 Reg.GetValueNames(ValueNames);
 for i := 0 to ValueNames.Count-1 do
  with ListView1.Items.Add do
   begin
   Caption := ValueNames[i];
   DataType := Reg.GetDataType(ValueNames[i]);
   Case DataType of
    rdString: s := Reg.ReadString(ValueNames[i]);
    rdInteger: s:= '0x'+IntToHex(Reg.ReadInteger(ValueNames[i]),8);
    rdBinary: s:='Binary';
   else s:= '???';
   end;
   SubItems.Add(s);
   ImageIndex :=1;
   end;
 ValueNames.Free;
end;
procedure TForm1.ShowSubKeys(ParentNode: TTreeNode;depth: Integer);
var ParentKey: string;
 KeyNames : TStringList;
 KeyInfo : TRegKeyInfo;
 CurNode : TTreeNode;
 i : Integer;
begin
 Cursor := crHourglass;
 TreeView1.Items.BeginUpdate;
 ParentKey := GetFullNodeName(ParentNode);
 if ParentKey<>'' then
  Reg.OpenKeyReadOnly(ParentKey)
 else
  Reg.OpenKeyReadOnly('\');
 Reg.GetKeyInfo(KeyInfo);
 if KeyInfo.NumSubKeys<=0 then Exit;
 KeyNames := TStringList.Create;
```

```
 Reg.GetKeyNames(KeyNames);
 While ParentNode.GetFirstChild<>nil do ParentNode.GetFirstChild.Delete;
 if (KeyNames.Count>0) then for i:=0 to KeyNames.Count-1 do
    begin
     Reg.OpenKeyReadOnly(ParentKey+'\'+KeyNames[i]);
     Reg.GetKeyInfo(KeyInfo);
     CurNode := TreeView1.Items.AddChild(ParentNode,KeyNames[i]);
     if KeyInfo.NumSubKeys>0 then
      begin
      TreeView1.Items.AddChild(CurNode,'');
      end;
    end;
 KeyNames.Free;
 TreeView1.Items.EndUpdate;
 Cursor := crDefault;
end;
procedure TForm1.TreeView1Expanded(Sender: TObject; Node: TTreeNode);
begin
 ShowSubKeys(Node,1);
end;

procedure TForm1.TreeView1GetImageIndex(Sender: TObject; Node:
TTreeNode);
begin
with Node do
begin
 if Expanded then ImageIndex := 2
 else ImageIndex := 3;
end;
end;
end.
```

Для работы с системным реестром выбран объект VCL `TRegistry`, удачно инкапсулирующий все предназначенные для этого функции Windows API. В обработчике события `OnCreate` главной формы создается объект `Reg`, а к списку `ListView1` добавляются два заголовка (свойство `Columns`).

Пояснений требует принцип построения дерева ключей. Во-первых, это приложение отображает только один из системных ключей (а именно `HKEY_LOCAL_MACHINE`); при желании его можно заменить или добавить остальные. Во-вторых, попытка построить все "развесистое" дерево ключей

сразу займет слишком много времени и наверняка не понравится пользователям. Вспомним, ведь утилита Registry Editor работает довольно быстро. Значит, придется строить дерево динамически — создавать и показывать дочерние узлы в момент развертывания родительского узла. Для этого используется событие `OnExpand` компонента `TreeView1`.

Остановимся на секунду. А какие узлы помечать кнопкой разворачивания (с пометкой "+"), ведь у родительского узла еще нет потомков? Выход из положения такой: в момент построения ключа проверить, есть ли у него дочерние. Если да, то к нему добавляется один (фиктивный) пустой ключ. Его единственная роль — дать системе поставить "+" против родительского узла.

Когда же пользователь щелкнул на кнопке, отмеченной знаком "+", и родительский узел разворачивается, фиктивный дочерний узел удаляется и вместо него создаются узлы настоящие, полученные путем сканирования реестра (см. метод `ShowSubKeys`).

Снабдим узлы картинками. Для этого в компонент `ImageList1` поместим картинки, соответствующие открытой и закрытой папкам. Напомним, что для отрисовки и смены картинок есть специальные события — `OnGetImageIndex` и `OnGetSelectedIndex`. В данном примере у двух этих событий один обработчик: развернутому узлу он сопоставляет картинку раскрытой папки, а свернутому — закрытой.

В заключение нужно сказать об очень важной особенности компонента `TListView`. Когда он отображает большой объем информации, обработка данных может затянуться очень и очень надолго, и занять слишком много памяти. Выход — перевести список в так называемый виртуальный режим. Он применяется для тех случаев, когда элементов в списке слишком много и хранить их там невозможно из соображений экономии времени или памяти.

Алгоритм действий прост.

1. Переводим компонент в виртуальный режим установкой свойства `OwnerData` в `True`.
2. Сообщаем списку, сколько в нем должно быть элементов, установкой нужного значения `Items.Count`.
3. Чтобы предоставить нужные данные, программист должен предусмотреть обработку событий `OnData`, `OnDataFind`, `OnDataHint` и `OnDataStateChange`. Как минимум нужно описать обработчик события `OnData`. `TLVOwnerDataEvent = procedure(Sender: TObject; Item: TListItem) of object;`

Вам передается объект `TListItem`, и внутри обработчика события `OnData` необходимо динамически "оформить" его (полностью — от заголовка до картинок).

Возникает это событие перед каждой перерисовкой списка. Так что, если сбор данных для вашего списка занимает более или менее продолжительное время, лучше не связывать его с событием `OnData`, т. к. перерисовка сильно затянется. К тому же, в виртуальном режиме сортировать список невозможно.

Календарь

Выбор даты — одна из часто встречающихся операций при вводе данных. Для облегчения этого действия предназначены два элемента управления. Компонент `TMonthCalendar` инкапсулирует календарь, панель которого содержит типовую таблицу на один месяц. Компонент `TDateTimePicker` совмещает календарь с однострочным текстовым редактором, позволяя вводить даты путем выбора из календаря.

Компонент *TMonthCalendar*

Этот элемент управления представляет собой панель с календарем на один месяц (рис. 8.7). Он обладает богатыми возможностями по настройке. Основные свойства компонента, отвечающие за внешний вид и управление календарем, представлены в табл. 8.5. Их назначение достаточно прозрачно и не требует комментариев.

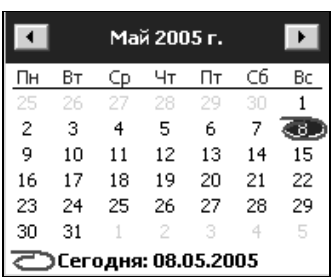


Рис. 8.7. Компонент `TMonthCalendar`

Сам календарь содержит в верхней части месяц и год, а расположенные слева и справа кнопки позволяют переходить к следующему и предыдущему месяцу. Красная окружность определяет текущую дату. Синий круг означает выбранную пользователем дату. При увеличении размеров в элементе управления отображается целое число календарей для месяцев, ближайших к текущему.

Таблица 8.5. *Основные свойства класса* `TMonthCalendar`

Объявление	Описание
`property CalColors: TMonthCalColors;`	Определяет цвета основных элементов календаря
`property Date: TDate;`	Содержит выбранную дату
`property EndDate: TDate;`	Содержит последнюю из выбранных дат при `MultiSelect` = `True`. Иначе совпадает со свойством `Date`
`type TCalDayOfWeek = (dowMonday, dowTuesday, dowWednesday, dowThursday, dowFriday, dowSaturday, dowSunday, dowLocaleDefault); property FirstDayOfWeek: TCalDayOfWeek;`	Определяет день, с которого начинается неделя. Значение по умолчанию `dowLocaleDefault` соответствует установкам ОС
`property MaxDate: TDate;`	Максимальная доступная для просмотра дата
`property MaxSelectRange: Integer;`	Максимальная доступная для выбора дата
`property MinDate: TDate;`	Минимальная доступная для просмотра дата
`property MultiSelect: Boolean;`	При значении `True` позволяет выбирать несколько дат одновременно
`property ShowToday: Boolean;`	Включает или отключает показ текущей даты в нижней части календаря
`property ShowTodayCircle: Boolean;`	Включает или отключает выделение текущей даты красным кругом
`property WeekNumbers: Boolean;`	Включает или отключает показ номеров недель в левой части календаря

Результат выбора даты в календаре сохраняется в свойстве `Date`. При установке выбора нескольких значений одновременно в свойстве `EndDate` содержится последняя дата, а в свойстве `Date` — самая ранняя из выбранных.

Метод-обработчик

```
property OnGetMonthInfo: TOnGetMonthInfoEvent;
```

вызывается при смене месяца.

Компонент *TDateTimePicker*

Безусловно, календарь будет очень полезен пользователям. Однако было бы желательно не только выбирать даты, но и вводить их в элементы управле-

ния. Компонент TDateTimePicker совмещает календарь и однострочный текстовый редактор, причем календарь полностью совпадает с рассмотренным выше (оба компонента являются наследниками класса TCommonCalendar). Свойства компонента приведены в табл. 8.6.

Компонент TDateTimePicker может обеспечивать ввод даты или времени.

Помимо календаря в элемент управления встроен флажок, который расположен в левой части редактора. Его видимостью можно управлять.

Таблица 8.6. *Основные свойства и методы класса* `TDateTimePicker`

Объявление	Описание
`type TDTCalAlignment = (dtaLeft, dtaRight);` `property CalAlignment: TDTCalAlignment;`	Выравнивает панель календаря по левой или правой стороне компонента
`property Checked: Boolean;`	Возвращает `True`, если флажок в редакторе включен
`type TDTDateFormat = (dfShort, dfLong);` `property DateFormat: TDTDateFormat;`	Определяет формат представления даты
`type TDTDateMode = (dmComboBox, dmUpDown);` `property DateMode: TDTDateMode;`	Задает стиль компонента
`property DroppedDown: Boolean;`	Возвращает `True`, если панель календаря включена
`type TDateTimeKind = (dtkDate, dtkTime);` `property Kind: TDateTimeKind;`	Определяет возвращаемый результат — дату или время. Время можно вводить только в стиле `dmUpDown`
`property ParseInput: Boolean;`	Включает или отключает метод-обработчик `OnUserInput`
`property ShowCheckbox: Boolean;`	Управляет видимостью флажка
`type TTime = type TDateTime;` `property Time: TTime;`	Содержит установленное время
`property OnChange: TNotifyEvent;`	Вызывается при вводе даты или времени
`property OnCloseUp: TNotifyEvent;`	Вызывается при сворачивании панели календаря

Таблица 8.6 *(окончание)*

Объявление	Описание
`property OnDropDown: TNotifyEvent;`	Вызывается при разворачивании панели календаря
`type TDTParseInputEvent = procedure(Sender: TObject; const UserString: string; var DateAndTime: TDateTime; var AllowChange: Boolean) of object;` `property OnUserInput: TDTParseInputEvent;`	Вызывается при прямом вводе значения в редактор. Параметр `UserString` содержит вводимое значение. Параметр `DateAndTime` содержит значение даты или времени. Параметр `AllowChange` управляет изменением значения

В зависимости от значения свойства `Kind` элемент управления настраивается на ввод даты или времени. Результат ввода даты сохраняется в свойстве `Date`. Дату можно выбирать из всплывающего календаря или путем перебора. Результат ввода времени сохраняется в свойстве `Time`.

Свойство `ParseInput` при значении `True` разрешает ручной ввод значения. В этом случае разработчик может использовать метод-обработчик

```
type TDTParseInputEvent = procedure(Sender: TObject; const UserString:
string; var DateAndTime: TDateTime; var AllowChange: Boolean) of object;
property OnUserInput: TDTParseInputEvent;
```

В нем можно предусмотреть необходимые действия, например, проверку введенного значения:

```
procedure TForm1.DateTimePicker1UserInput(Sender: TObject;
  const UserString: String; var DateAndTime: TDateTime;
  var AllowChange: Boolean);
begin
 try
  DateAndTime := StrToDateTime(UserString);
 except
  on E: EConvertError do ShowMessage('Неверное значение');
 end;
end;
```

Обратите внимание, что здесь обязательно должно присутствовать присвоение результата ввода параметру `DateAndTime`, иначе элемент управления не получит новое значение.

Панель состояния *TStatusBar*

Этот вид элементов управления применяется уже достаточно давно. Его роль заключается в отображении различного рода справочной информации. Панель состояния инкапсулирована в компоненте `TStatusBar`.

Обычно панель состояния размещается в нижней части окна. Поэтому при переносе на форму свойство `Align` всегда имеет значение `alBottom`. Панель состояния можно разделить на произвольное число самостоятельных частей. Каждая такая часть описывается объектом `TStatusPanel`. Коллекция всех таких объектов находится в свойстве

```
property Panels: TStatusPanels;
```

Например, для того, чтобы показать в панели состояния текущую дату и время, в методе-обработчике `OnTimer` компонента `TTimer` достаточно предусмотреть следующий код:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
 StatusBar1.Panels[0].Text := DateToStr(Now);
 StatusBar1.Panels[1].Text := TimeToStr(Now);
end;
```

Впрочем, панель состояния можно сделать сплошной. Для этого свойство `SimplePanel` должно иметь значение `True`. В этом случае текст панели должен содержаться в свойстве `SimpleText`.

Расширенный комбинированный список *TComboBoxEx*

Что отличает этот "продвинутый" выпадающий список от обычного `TComboBox`? С функциональной точки зрения основных отличий два: возможность добавлять картинки к элементам и выравнивать последние с разным отступом, имитируя иерархию.

Реализовано это следующим образом.

У компонента `TComboBoxEx`, помимо свойства `Items`, есть свойство

```
property ItemsEx: TComboExItems;
```

которое представляет собой коллекцию элементов типа `TComboExItem`. Щелкнув на этом свойстве в Инспекторе объектов, увидим типичный редактор коллекций, где каждый элемент обладает такими опубликованными свойствами:

- `Caption` — отвечает за заголовок элемента, каким он буден виден в списке;

- `Data` — это нетипизированный указатель на прикрепляемые к элементу данные;
- `Indent` — задает отступ от левого края списка. В документации написано, что оно задается в пикселах. Это не совсем так — на самом деле одна единица значения свойства соответствует десятку пикселов;
- `ImageIndex` — обычный номер картинки;
- `SelectedImageIndex` — номер для выбранного элемента;
- `OverlayImageIndex` — задает номер картинки, являющейся накладываемой маской для первых двух. Она должна быть черно-белой: белые области прозрачны для исходной картинки, черные — нет. Все три индекса указывают на один и тот же список картинок, задаваемый свойством `images` родительского компонента.

Дополнительные опции в расширенном выпадающем списке задаются свойством `StyleEx`. Это — множество из четырех флагов, установка которых сводится к разрешению или запрету перечисленных новых свойств.

Создание нового компонента на базе элементов управления из библиотеки ComCtl32

С каждой версией Internet Explorer Microsoft заново поставляет библиотеку COMCTL32 с новыми элементами управления. Программисты Borland пытаются поспеть за ними, но получается это не всегда. Так что полезно было бы и самому научиться создавать оболочку для новых и необходимых элементов управления, тем более, что это несложно. Рассмотрим это на примере.

Подходящей кандидатурой может служить редактор IP-адресов. Это элемент, упрощающий редактирование адресов для многих Internet-компонентов и приложений.

Мастер создания новых компонентов создаст для нас шаблон. Он открывается при выборе команды **New VCL Component** меню **Component**. На первой странице мастера New VCL Component необходимо выбрать программную платформу (рис. 8.8). В нашем случае это Win32.

На следующей странице нужно указать класс-предок создаваемого компонента (рис. 8.9). Поскольку элементы из состава COMCTL32 есть ни что иное, как окна со специфическими свойствами, наш компонент мы породим от `TWinControl`. IP-редактор представляет собой окно класса `WC_IPADDRESS`.

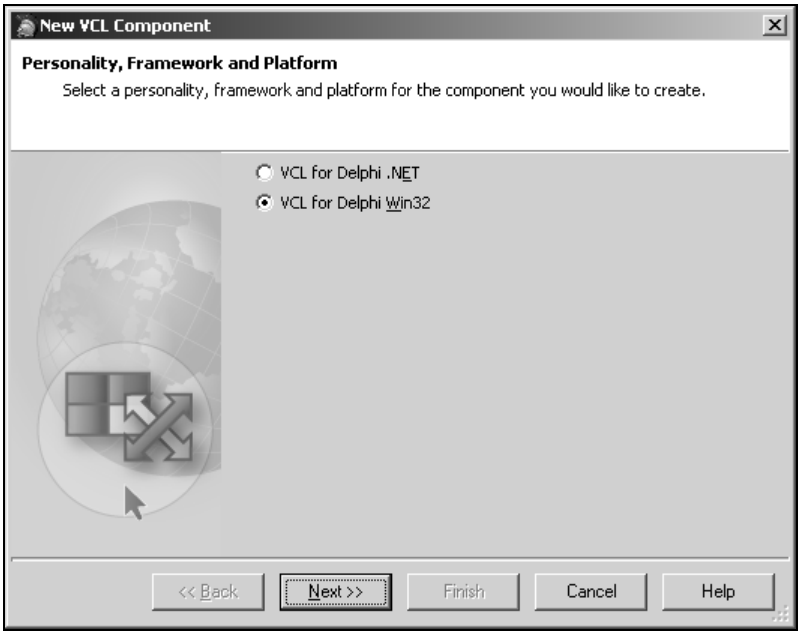


Рис. 8.8. Первая страница мастера создания новых компонентов

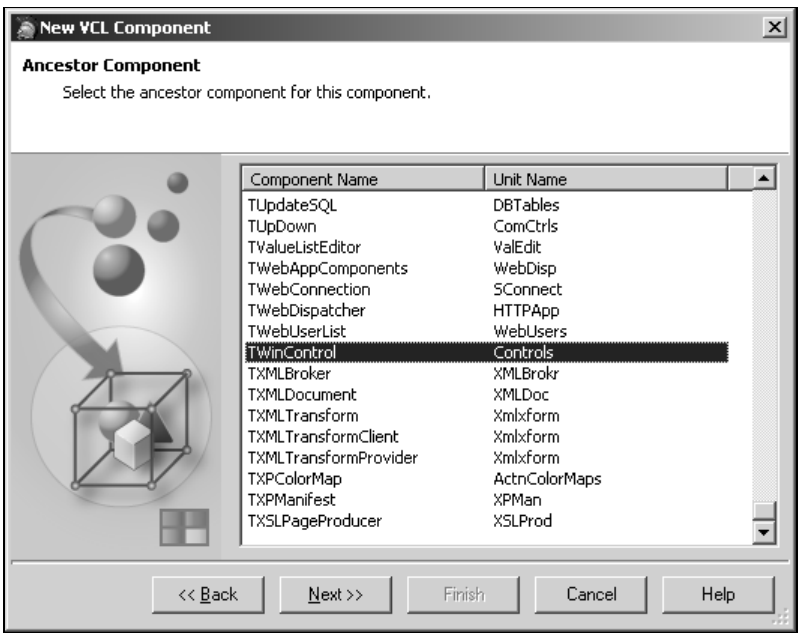


Рис. 8.9. Выбор класса-предка в мастере создания новых компонентов

Затем следует задать имя компонента, местоположение его модулей и категорию Палитры инструментов, где компонент будет располагаться (рис. 8.10). Название нового компонента выбрано `TCustomIPEdit`. Такая схема принята разработчиками Delphi для большинства компонентов VCL. Непосредственным предком, допустим, `TEdit` является компонент `TCustomEdit`. Так поступим и мы.

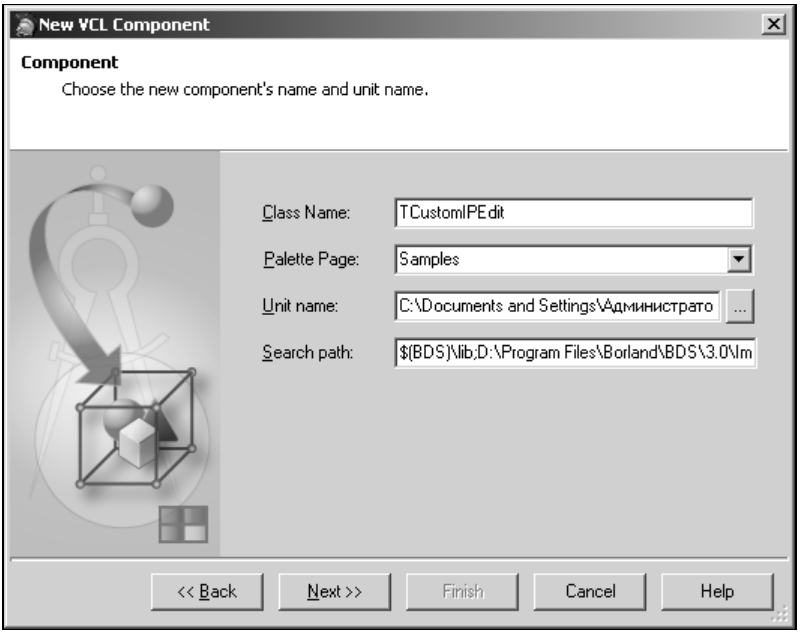


Рис. 8.10. Выбор имени компонента в мастере создания новых компонентов

И на последней странице мастера создания компонентов нужно указать, требуется ли создавать новый модуль для компонента или компонент будет добавлен из существующего пакета. В нашем случае модуль находится в папке примера под именем uIPEdit.pas (листинг 8.4).

Первым делом при создании компонента, особо не раздумывая, следует опубликовать типовые свойства и события, которые есть у большинства визуальных компонентов. Чтобы не занимать место в книге, позаимствуем их список у любого другого компонента из COMCTRLS.PAS.

Далее приступим к описанию свойств, которыми будет отличаться наш компонент от других. Возможности, предоставляемые IP-редактором, описаны в справочной системе MSDN. Визуально он состоит из четырех полей, разделенных точками (рис. 8.11).

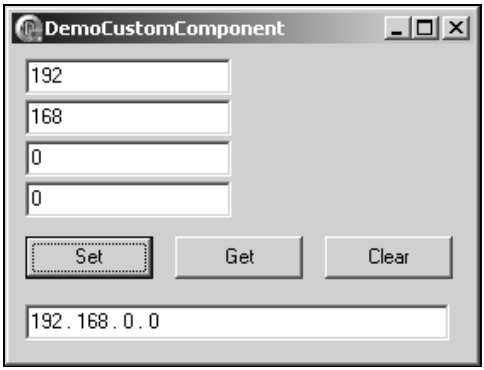


Рис. 8.11. Тестовое приложение, содержащее IP-редактор (внизу)

Для каждого из полей можно задать отдельно верхнюю и нижнюю границы допустимых значений. Это удобно, если вы планируете работать с адресами какой-либо конкретной IP-сети. По умолчанию границы равны 0..255.

Элемент управления обрабатывает шесть сообщений (см. документацию MSDN), которые сведены в табл. 8.7.

Таблица 8.7. *Сообщения, обрабатываемые элементом управления IP Address Control*

Сообщение	Назначение
`IPM_CLEARADDRESS`	Очистить поле адреса
`IPM_GETADDRESS`	Считать адрес
`IPM_ISBLANK`	Проверить, не пустое ли поле адреса
`IPM_SETADDRESS`	Установить адрес
`IPM_SETFOCUS`	Передать фокус заданному полю элемента управления
`IPM_SETRANGE`	Установить ограничения на значения в заданном поле

Кроме перечисленных, IP-редактор извещает приложение об изменениях, произведенных пользователем, путем посылки ему сообщения `WM_NOTIFY`.

Следует иметь в виду, что IP-редактор не является потомком обычного редактора (`TCustomEdit`) и не обрабатывает характерные для того сообщения `EM_XXXX`, так что название `TCustomIPEdit` отражает только внешнее сходство.

В создаваемом коде компонента (листинг 8.3) первым делом нужно переписать конструктор `Create` и метод `CreateParams`. Последний вызывается перед созданием окна для установки его будущих параметров. Именно здесь

нужно инициализировать библиотеку общих элементов управления COMCTL32 и породить новый класс окна.

Листинг 8.3. Код создаваемого компонента

```
constructor TIPEditor.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := [csCaptureMouse, csClickEvents, csDoubleClicks,
csOpaque];
  Color := clBtnFace;
  Width := 160;
  Height := 25;
  Align := alNone;
end;
 procedure TIPEditor.CreateParams(var Params: TCreateParams);
 begin
  InitCommonControl(ICC_INTERNET_CLASSES);
  inherited CreateParams(Params);
  CreateSubClass(Params, WC_IPADDRESS);
 end;
```

После создания свое значение получает дескриптор окна `Handle` (это свойство унаследовано от `TWinControl`). Все чтение/запись свойств элемента происходит путем обмена сообщениями с использованием этого дескриптора. Минимально необходимыми для работы являются свойства IP (задает IP-адрес в редакторе), `IPString` (отображает его в виде текстовой строки) и процедура `Clear` (очищает редактор). Листинг 8.4 иллюстрирует это.

Листинг 8.4. Исходный код компонента `TCustomIPEdit`

```
unit uIPEdit;
interface
uses
  Windows, Messages, SysUtils, Classes, Controls;
type
  TCustomIPEdit = class(TWinControl)
  private
    { Private declarations }
   FIPAddress: DWORD;
```

```
  FIPLimits: array [0..3] of word;
  FCurrentField : Integer;
  //procedure CMWantSpecialKey(var Msg: TCMWantSpecialKey); message
  CM_WANTSPECIALKEY;
  procedure WMGetDlgCode(var Message: TWMGetDlgCode);message
  WM_GETDLGCODE;
  procedure CMDialogChar(var Message: TCMDialogChar); message
  CM_DIALOGCHAR;
  //procedure CMDialogKey(var Message: TCMDialogKey); message
  CM_DIALOGKEY;
  procedure CNNotify(var Message: TWMNotify); message CN_NOTIFY;
 protected
  { Protected declarations }
  function GetIP(Index: Integer): Byte;
  procedure SetIP(Index: Integer; Value: Byte);
  function GetMinIP(Index: Integer): Byte;
  procedure SetMinIP(Index: Integer; Value: Byte);
  function GetMaxIP(Index: Integer): Byte;
  procedure SetMaxIP(Index: Integer; Value: Byte);
  function GetIPString: string;
  procedure SetIPString( Value: string );
  function IsBlank: boolean;
  procedure SetCurrentField( Index: Integer );
  //
  procedure CreateParams(var Params: TCreateParams); override;
  procedure CreateWnd; override;
  //procedure KeyDown(var Key: Word; Shift: TShiftState);override;
  function IPDwordToString( dw: DWORD ): string;
  function IPStringToDword( s: string ): DWORD;
 public
   { Public declarations }
  constructor Create(AOwner: TComponent); override;
  property IP[Index: Integer]: byte read GetIP write SetIP;
  property MinIP[Index: Integer]: byte read GetMinIP write SetMinIP;
  property MaxIP[Index: Integer]: byte read GetMaxIP write SetMaxIP;
  property IPString : string read GetIPString write SetIPString;
  property CurrentField : Integer read FCurrentField write
  SetCurrentField;
  procedure Clear;

 end;
```

```
 TIPEdit = class(TCustomIPEdit)
 published
    property Align;
    property Anchors;
    property BorderWidth;
    property DragCursor;
    property DragKind;
    property DragMode;
    property Enabled;
    property Font;
    property Hint;
    property Constraints;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property TabOrder;
    property TabStop;
    property Visible;
    property OnContextPopup;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDock;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDock;
    property OnStartDrag;
    { Published declarations }
    property IPString;
 end;
procedure Register;
implementation
uses Graphics, commctrl, comctrls;
constructor TCustomIPEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
```

```
  FIPAddress := 0;
  ControlStyle := [csCaptureMouse, csClickEvents, csDoubleClicks,
csOpaque];
  Color := clBtnFace;
  Width := 160;
  Height := 25;
  Align := alNone;
  TabStop := True;
end;
procedure TCustomIPEdit.CreateParams(var Params: TCreateParams);
begin
  InitCommonControl(ICC_INTERNET_CLASSES);
  inherited CreateParams(Params);
  CreateSubClass(Params, WC_IPADDRESS);
  with Params do
  begin
    Style := WS_VISIBLE or WS_BORDER or WS_CHILD;
    if NewStyleControls and Ctl3D then
    begin
      Style := Style and not WS_BORDER;
      ExStyle := ExStyle or WS_EX_CLIENTEDGE;
    end;
  end;
end;
procedure TCustomIPEdit.CreateWnd;
var i: Integer;
begin
 inherited CreateWnd;
 Clear;
{ for i := 0 to 3 do
  begin
   MinIP[i] := 0;
   MaxIP[i] := $FF;
  end;}
 CurrentField := 0;
end;
procedure TCustomIPEdit.WMGetDlgCode(var Message: TWMGetDlgCode);
begin
  inherited;
```

```
  Message.Result := {Message.Result or} DLGC_WANTTAB;
end;
procedure TCustomIPEdit.CNNotify(var Message: TWMNotify);
begin
  with Message.NMHdr^ do begin
    case Code of
      IPN_FIELDCHANGED :
        begin
          FCurrentField := PNMIPAddress( Message.NMHdr )^.iField;
          {if Assigned(OnIpFieldChange) then
            with PNMIPAdress(Message.NMHdr)^ do begin
               OnIPFieldChange(Self, iField, iValue);}
        end;
    end;
  end;
end;
{procedure TCustomIPEdit.KeyDown(var Key: Word; Shift: TShiftState);
begin
 inherited KeyDown( Key, Shift );
 if Key = VK_TAB then
  if ssShift in Shift then
     CurrentField := (CurrentField - 1 + 4) mod 4
    else
     CurrentField := (CurrentField + 1) mod 4;
end;}
{procedure TCustomIPEdit.CMWantSpecialKey(var Msg: TCMWantSpecialKey);
begin
  inherited;
  //Msg.Result := Ord(Char(Msg.CharCode) = #9);
end;}
procedure TCustomIPEdit.CMDialogChar(var Message: TCMDialogChar);
begin
 with Message do
   if CharCode = VK_TAB then
    begin
     Message.Result := 0;
     if GetKeyState(VK_SHIFT)<>0 then
      begin
       if (CurrentField=0) then Exit;
```

```
       CurrentField := CurrentField - 1;
      end
    else
      begin
       if (CurrentField=3) then Exit;
       CurrentField := CurrentField + 1;
      end;
    Message.Result := 1;
    end //VK_TAB
   else
     inherited;
end;
{procedure TCustomIPEdit.CMDialogKey(var Message: TCMDialogKey);
begin
  if (Focused or Windows.IsChild(Handle, Windows.GetFocus)) and
    (Message.CharCode = VK_TAB) and (GetKeyState(VK_CONTROL) < 0) then
   begin
    if GetKeyState(VK_SHIFT)<>0 then
     CurrentField := (CurrentField - 1 + 4) mod 4
    else
     CurrentField := (CurrentField + 1) mod 4;
    Message.Result := 1;
   end
 else
  inherited;
end;}
 function TCustomIPEdit.GetIP(Index: Integer): Byte;
 begin
  SendMessage( Handle,IPM_GETADDRESS,0,longint(@FipAddress));
  case Index of
   1 : Result := FIRST_IPADDRESS(FipAddress);
   2 : Result := SECOND_IPADDRESS(FipAddress);
   3 : Result := THIRD_IPADDRESS(FipAddress);
   4 : Result := FOURTH_IPADDRESS(FipAddress);
  else Result := 0;
  end;
 end;
```

```
procedure TCustomIPEdit.SetIP(Index: Integer; Value: Byte);
begin
 case Index of
  1: FIPAddress := FIPAddress AND $FFFFFF or DWORD(Value) shl 24;
  2: FIPAddress := FIPAddress AND $FF00FFFF or DWORD(Value) shl 16;
  3: FIPAddress := FIPAddress AND $FFFF00FF or DWORD(Value) shl 8;
  4: FIPAddress := FIPAddress AND $FFFFFF00 or DWORD(Value);
  else Exit;
 end;
 SendMessage( Handle, IPM_SETADDRESS, 0, FIPAddress );
end;
function TCustomIPEdit.GetMinIP(Index: Integer): Byte;
begin
 if (Index<0) or (Index>3) then
  Result := 0
 else
  Result := LoByte( FIPLimits[Index] );
end;
procedure TCustomIPEdit.SetMinIP(Index: Integer; Value: Byte);
begin
 if (Index<0) or (Index>3) then Exit;
 FIPLimits[Index] := MAKEIPRANGE( HiByte(FIPLimits[Index]), Value );
 SendMessage( Handle, IPM_SETRANGE, Index, FIPLimits[Index] );
end;
function TCustomIPEdit.GetMaxIP(Index: Integer): Byte;
begin
 if (Index<0) or (Index>3) then
  Result := 0
 else
  Result := HiByte( FIPLimits[Index] );
end;
procedure TCustomIPEdit.SetMaxIP(Index: Integer; Value: Byte);
begin
 if (Index<0) or (Index>3) then Exit;
 FIPLimits[Index] := MAKEIPRANGE( Value, LoByte(FIPLimits[Index]) );
 SendMessage( Handle, IPM_SETRANGE, Index, FIPLimits[Index] );
end;
```

```
procedure TCustomIPEdit.Clear;
begin
 SendMessage( Handle, IPM_CLEARADDRESS, 0, 0 );
end;
function TCustomIPEdit.IsBlank: boolean;
begin
 Result:= SendMessage( Handle, IPM_ISBLANK, 0, 0 ) = 0;
end;
procedure TCustomIPEdit.SetCurrentField( Index: Integer );
begin
 if (Index<0) or (Index>3) then Exit;
 FCurrentField := Index;
 SendMessage( Handle, IPM_SETFOCUS, wParam(FCurrentField), 0 );
end;
function TCustomIPEdit.IPDwordToString( dw: DWORD ): string;
begin
  Result := Format('%d.%d.%d.%d',
  [FIRST_IPADDRESS( dw ),
  SECOND_IPADDRESS( dw ),
  THIRD_IPADDRESS( dw ),
  FOURTH_IPADDRESS( dw )]);
end;
function TCustomIPEdit.IPStringToDword( s: string ): DWORD;
var i,j : Integer;
 NewAddr, Part : DWORD;
begin
NewAddr := 0;
try
i := 0;
 repeat
  j := Pos( '.', s );
  if j<=1 then
   if i<3 then
    Abort
   else
    Part := StrToInt( s )
  else
   Part := StrToInt( Copy( s, 1, j-1 ) );
```

```
   if Part>255 then Abort;
   Delete( s, 1, j );
   NewAddr := (NewAddr shl 8) or Part;
   Inc(i);
  until i>3;
  Result := NewAddr;
  //Windows.MessageBox( 0, pChar(IntToHex(FIPAddress, 8 )), '', MB_Ok );
 except
 end;
 end;
 function TCustomIPEdit.GetIPString: string;
 begin
  SendMessage( Handle,IPM_GETADDRESS,0, longint(@FIPAddress) );
  Result := IpDwordToString( FIPAddress );
 end;
 procedure TCustomIPEdit.SetIPString( Value: string );
 begin
  FIPAddress := IPStringToDword( Value );
  SendMessage( Handle, IPM_SETADDRESS, 0, FIPAddress);
 end;
 procedure Register;
 begin
  RegisterComponents('Samples', [TIPEdit]);
 end;
end.
```

Для удобства полезно было бы добавить к этому компоненту задание диапазона для каждого из четырех составляющих и средства преобразования текстовой строки в двоичный адрес. Но это уже совсем другая история, к библиотеке ComCtl32 отношения не имеющая.

Пользовательский интерфейс Windows XP

При первом знакомстве с Windows XP сразу же обращаешь внимание на существенные изменения в оформлении пользовательского интерфейса этой операционной системы. Все нововведения основаны на использовании нового способа отрисовки элементов управления, который реализован в системной динамической библиотеке ComCtl32.dll версии 6. Эта библиотека входит в состав операционной системы Windows XP.

Примечание

Обратите внимание, что все более ранние операционные системы фирмы Microsoft используют ComCtl32.dll пятой версии с традиционным стилем оформления элементов управления. И, конечно, нельзя заставить Windows 2000 выглядеть как Windows XP простой заменой версий ComCtl32.dll. Операционная система должна уметь работать с новыми возможностями.

Теперь элементы управления в стиле Windows XP могут иметь собственную цветовую палитру, текстуры, стили заполнения и, самое главное, различные методы отрисовки своих составных частей. Например, элемент управления может рисовать собственное поле ввода данных обычным способом, а рамку — с использованием маски XOR.

Совокупность правил и методов поведения, оформления и отрисовки элементов управления называется *визуальным стилем* (visual style). Совместно с Windows XP поставляется один готовый стиль.

Если разработчик хочет, чтобы его приложение могло задействовать возможности Windows XP, он должен включить в состав проекта специальный ресурс — *манифест* (manifest). Манифест представляет собой документ XML, содержащий необходимую для ComCtl32.dll версии 6 информацию.

Такое приложение будет "чувствовать себя как дома" в среде Windows XP. Но и при запуске в старых операционных системах со стандартным набором возможностей ComCtl32.dll пятой версии оно будет выглядеть как и любое другое приложение.

Кроме этого, Windows XP также имеет библиотеку ComCtl32.dll пятой версии, и именно она установлена по умолчанию.

Манифест Windows XP

Итак, начнем с манифеста. Он представляет собой документ в формате XML, содержащий всю информацию, необходимую для взаимодействия приложения и библиотеки ComCtl32.dll шестой версии.

Примечание

Следует отметить, что манифесты широко используются во многих продуктах и технологиях, работающих в операционных системах Microsoft. С полной схемой XML манифеста вы можете ознакомиться в документации Microsoft MSDN.

Пример манифеста представлен в листинге 8.5.

Листинг 8.5. Манифест Windows XP

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
```

```
<assemblyIdentity
version="1.0.0.0"
processorArchitecture="X86"
name="CompanyName.ProductName.YourApp"
type="win32"
/>
<description>Your application description here.</description>
<dependency>
<dependentAssembly>
<assemblyIdentity
type="win32"
name="Microsoft.Windows.Common-Controls"
version="6.0.0.0"
processorArchitecture="X86"
publicKeyToken="6595b64144ccf1df"
language="*"
/>
</dependentAssembly>
</dependency>
</assembly>
```

Обратите внимание, что в элементе

```
<dependency>
       <dependentAssembly>
       <assemblyIdentity>
```

описывается версия системной библиотеки, используемой приложением для отрисовки элементов управления.

При загрузке приложения операционная система Windows XP считывает манифест (если он есть) и получает информацию о том, что для выполнения приложения потребуется библиотека ComCtl32.dll шестой версии.

Помимо этой информации, манифест может содержать и другие необходимые сведения о приложении и условиях его работы. Например, общая информация о приложении и его версии представлены элементом `<assemblyIdentity>`.

Вы можете добавить манифест в ваше приложение двумя способами.

- С помощью компонента `TXPManifest`.
- Включить в ресурсы приложения вручную.

Рассмотрим их подробнее.

Компонент *TXPManifest*

В категории Win32 Палитры инструментов Delphi имеется компонент `TXPManifest`. Будучи добавленным к проекту, он обеспечивает компиляцию манифеста Windows XP в исполняемый файл приложения. Основой является стандартный манифест Delphi для Windows XP, содержащийся в файле ресурсов ..\Lib\WindowsXP.res (листинг 8.6).

Листинг 8.6. Манифест Delphi для Windows XP

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity
       type="win32"
    name="DelphiApplication"
    version="3.2.0.0"
       processorArchitecture="*"/>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        publicKeyToken="6595b64144ccf1df"
        language="*"
        processorArchitecture="*"/>
    </dependentAssembly>
  </dependency>
</assembly>
```

К сожалению, версия программного продукта (ProductVersion), а также любая другая информация о версии, содержащаяся в проекте (файлы BDSPROJ и RES) и настраиваемая в диалоге **Project Options** среды разработки Delphi, никак не влияет на содержимое манифеста. Поэтому при настройке манифеста в соответствии с потребностями приложения вам придется предварительно отредактировать файл WindowsXP.res или поправить манифест прямо в исполняемом файле. (Ввиду частых перекомпиляций проекта второй вариант представляется довольно обременительным.)

Включение манифеста Windows XP в ресурсы приложения

Так как использование стандартного компонента `TXPManifest` требует настройки исходного манифеста Delphi для каждого проекта, было бы неплохо изыскать более удобный способ. В качестве альтернативы вы можете прикомпилировать манифест к файлу вашего проекта и по мере надобности редактировать его, не опасаясь, что ваша забывчивость может отразиться на версиях в манифестах других приложений.

Для начала необходимо создать исходный файл ресурса RC, включающий единственную строку:

```
1 24 "XP.manifest"
```

Здесь

- 1 — номер ресурса версии библиотеки ComCtl32.dll;
- 24 — номер ресурса манифеста (нумерация соответствует заголовочным файлам, распространяемых Microsoft);
- `"XP.manifest"` — имя файла с документом XML, содержащим манифест.

Естественно, манифест нужно настроить в соответствии с потребностями вашего проекта.

Теперь нужно откомпилировать файл ресурса при помощи строчного компилятора ресурсов ..bin\brcc32.exe и разместить его в папке проекта.

И последняя операция — добавьте в исходный код файла проекта директиву подключения ресурса манифеста:

```
{$R xpmanifest.res}
```

В результате при компиляции проекта манифест из ресурса будет добавлен в исполняемый файл приложения.

Пример ресурсов манифеста имеется на компакт-диске, прилагаемом к этой книге.

Визуальные стили и темы оформления

Теперь давайте более подробно разберемся с визуальными стилями и их влиянием на пользовательский интерфейс приложений.

Начиная с операционной системы Widows 95 пользователям был доступен пакет обновления Microsoft Plus!, который позволял использовать темы оформления рабочего стола Windows. *Темой* называется совокупность настроек цветов, шрифтов, курсоров и прочих ресурсов, необходимых для создания унифицированного пользовательского интерфейса.

Все параметры одной темы сохраняются в файле с расширением theme в виде секций и значений, подобно файлам INI. Существующие темы доступны для выбора в системном диалоге **Display Options**.

Визуальные стили, интегрированные в Windows XP, управляют внешним видом и поведением элементов управления. При этом визуальный стиль использует настройки параметров пользовательского интерфейса, заданные текущей темой. Для управления темами визуального стиля операционная система использует *менеджер тем*.

Визуальный стиль позволяет настраивать внешний вид не только элементов управления в целом, но и его составных частей. Правила и методы отрисовки сохраняются в файле с расширением mst, который входит в состав визуального стиля.

Совместно с Windows XP поставляется только один визуальный стиль, и он производит приятное и свежее впечатление о пользовательском интерфейсе операционной системы.

Визуальные стили в Delphi

В *главе 9* мы детально рассмотрим роль *действий* при разработке пользовательского интерфейса приложения и специальном компоненте для управления действиями — `TActionManager`. Немного забегая вперед скажем, что этот компонент является своего рода "командным пунктом", из которого должны управляться элементы управления приложения. Сейчас же нас интересует только одно свойство этого компонента:

```
property Style: TActionBarStyle;
```

По умолчанию среда разработки Delphi предлагает два стиля.

- `Standard` — приложение использует системную библиотеку ComCtl32.dll пятой версии.
- `Windows XP` — приложение использует системную библиотеку ComCtl32.dll шестой версии и единственный стандартный визуальный стиль Windows XP.

Эти стили применимы только к элементам управления, размещенным на панелях инструментов (`TActionToolBar`), созданных в компоненте `TActionManager`.

Однако не торопитесь возмущаться явной ограниченностью выбора. Вы можете создать собственный стиль самостоятельно. Правда, это потребует очень много усилий — ведь на основе базовых классов элементов управления вам потребуется создать собственные классы с нужными вам поведением и внешним видом.

Для этого необходимо создать класс нового визуального стиля на основе класса `TActionBarStyleEx`. Затем новый стиль регистрируется при помощи процедуры

```
procedure RegisterActnBarStyle(AStyle: TActionBarStyleEx);
```

После этого ваш стиль становится доступным для свойства `Style` компонента `TActionManager`.

Чтобы разрегистрировать стиль, выберите процедуру

```
procedure UnRegisterActnBarStyle(AStyle: TActionBarStyleEx);
```

Например, обе эти операции удобно выполнить при инициализации и деинициализации модуля, описывающего класс стиля (листинг 8.7).

Листинг 8.7. Вариант регистрации и разрегистрации собственного визуального стиля

```
var MyStyle: TMyStyleActionBars;
...
initialization
  MyStyle := TMyStyleActionBars.Create;
  RegisterActnBarStyle(MyStyle);
finalization
  UnregisterActnBarStyle(MyStyle);
  MyStyle.Free;
end.
```

Изменить стиль приложения можно с помощью глобальной переменной нового стиля (см. листинг 8.7). Ее достаточно присвоить свойству `Style`:

```
ActionManager1.Style := MyStyle;
```

При смене стиля все элементы управления, расположенные на панелях компонента `ActionManager1`, будут уничтожены и созданы заново с использованием настроек нового стиля.

Класс `TActionBarStyleEx` имеет всего несколько методов, которые необходимо перекрыть при создании собственного стиля. Все они возвращают классы объектов, используемых при создании пользовательского интерфейса. Рассмотрим их

Функция

```
function GetStyleName: string;
```

возвращает имя стиля.

Функция

```
function GetColorMapClass(ActionBar: TCustomActionBar):
TCustomColorMapClass;
```

позволяет получить ссылку на класс компонента настройки цветовой палитры *(см. ниже в этой главе "Компоненты настройки цветовой палитры")*, используемый панелью инструментов.

Следующие три метода дают информацию о классах различных типов элементов управления, используемых при проектировании пользовательского интерфейса с помощью компонента `TActionManager`.

Для того чтобы получить доступ к элементам управления, связанным со стилем, предназначен метод

```
function GetControlClass(ActionBar: TCustomActionBar; AnItem:
TActionClientItem): TCustomActionControlClass;
```

Он возвращает класс элемента управления из панели `ActionBar`, связанного с элементом управления `AnItem`. Именно эта функция вызывается при создании элементов управления в панелях инструментов компонента `TActionManager`.

Как уже говорилось, при присвоении свойству `Style` компонента `TActionManager` нового значения (экземпляра класса разработанного вами визуального стиля) уничтожаются все существующие элементы управления и затем создаются новые. И в процессе создания каждого визуального компонента вызывается функция `GetControlClass` нового стиля, а возвращенное ею значение используется для вызова конструктора соответствующего класса.

Аналогично, для получения класса в панели меню применяется метод

```
function GetPopupClass(ActionBar: TCustomActionBar): TGetPopupClass;
```

и для классов кнопок панели инструментов — функция

```
function GetScrollBtnClass: TCustomToolScrollBtnClass;
```

А класс самой панели инструментов возвращает функция

```
function GetAddRemoveItemClass(ActionBar: TCustomActionBar):
TCustomAddRemoveItemClass;
```

Итак, после разработки и отладки собственных классов элементов управления с заданными свойствами вам потребуется создать потомка от класса `TActionBarStyleEx` и перекрыть все перечисленные функции так, чтобы они возвращали нужные классы для выбранных типов элементов управления.

Theme API

Помимо описанного способа создания и управления визуальными стилями в распоряжении программиста есть функции Theme API, разработанные Microsoft для этих целей.

Для того чтобы задействовать Theme API, можно стандартным способом подключить к проекту динамическую библиотеку theme.dll:

```
var ThemeDLL: HINST;
...
begin
 ThemeDLL := LoadLibrary('theme.dll');
 if ThemeDLL <> 0 then
 begin
  ...
 end;
end;
```

После этого возможности данного программного интерфейса доступны напрямую. С деталями его реализации вы же можете ознакомиться в документации Microsoft MSDN.

Однако можно поступить проще. В составе Delphi имеется модуль UxTheme.pas, в котором как раз и реализованы возможности Theme API. Кроме этого, модуль Themes.pas содержит классы для основных элементов управления, которые могут использоваться при создании визуальных стилей, а также класс менеджера тем `TThemeServices`.

Так как детальное обсуждение возможностей Theme API выходит за рамки этой книги, в листинге 8.8 представлен схематичный пример использования функций этого программного интерфейса. Кроме того, как и все остальные API, работающие с GUI (Graphic User Interface) операционной системы, реальный код с функциями Theme API всегда перегружен многочисленными и "ужасными" на вид (а на самом деле вполне безобидными) функциями, рассчитывающими области вывода, неклиентские зоны оконных элементов и т. д.

Поэтому наш пример лишь показывает, как загрузить динамическую библиотеку theme.dll и получить ссылку на тему визуального стиля для текущего окна и кнопочного элемента управления.

Листинг 8.8. Пример использования функций Theme API в Delphi

```
var DC: HDC;
    CurrentThemeData: HTHEME;
```

```
begin
 if UseThemes and InitThemeLibrary then
 try
   DC := GetWindowDC(Handle);
   try
    CurrentThemeData := OpenThemeData(0, 'button');
    CloseThemeData(CurrentThemeData);
   finally
    ReleaseDC(Handle, DC);
   end
 finally
  FreeThemeLibrary;
 end
 else
  ShowMessage('Приложение или операционная система не поддерживают
  использование Theme API');
end;
```

Функция

```
function UseThemes: Boolean;
```

проверяет способность операционной системы и приложения использовать Theme API.

Методы

```
function InitThemeLibrary: Boolean;
procedure FreeThemeLibrary;
```

соответственно инициализируют и выгружают библиотеку theme.dll.

Графический контекст `DC` наверняка понадобится при отрисовке элементов управления *(см. главу 12)*.

Функция

```
OpenThemeData: function(hwnd: HWND; pszClassList: LPCWSTR): HTHEME;
stdcall;
```

возвращает дескриптор темы для текущего визуально стиля и класса, имя которого представлено параметром `pszClassList`.

После того как тема загружена, становятся доступными разнообразные графические функции, входящие в состав программного интерфейса.

При завершении работы не забудьте освободить занятые дескрипторы графического контекста и темы. Для темы используйте функцию

```
CloseThemeData: function(hTheme: HTHEME): HRESULT; stdcall;
```

Заинтересованный читатель найдет подробное описание нужных функций Theme API в Microsoft MSDN или же может полюбопытствовать содержимым модулей UxTheme.pas и Themes.pas.

Компоненты настройки цветовой палитры

Помимо создания собственных визуальных стилей, что является делом довольно трудоемким и хлопотным, вы можете изменить внешний вид пользовательского интерфейса приложения более легким способом. Впервые в составе Палитры инструментов Delphi есть специализированные компоненты, позволяющие настраивать цветовую палитру всех возможных деталей пользовательского интерфейса одновременно. Эти компоненты расположены в категории **Additional**.

- `TStandardColorMap` — по умолчанию настроен на стандартную цветовую палитру Windows.
- `TXPColorMap` — по умолчанию настроен на стандартную цветовую палитру Windows XP.
- `TTwilightColorMap` — по умолчанию настроен на стандартную полутоновую (черно-белую) палитру Windows.

Все они представляют собой контейнер, содержащий цвета для раскраски различных деталей элементов управления. Разработчику необходимо лишь настроить эту цветовую палитру и по мере необходимости подключать к пользовательскому интерфейсу приложения. Для этого снова необходим компонент `TActionManager`.

Все панели инструментов (класс `TActionToolBar`), созданные в этом компоненте *(см. главу 9)*, имеют свойство

```
property ColorMap: TCustomActionBarColorMap;
```

в котором и задается необходимый компонент цветовой палитры. Сразу после подключения все элементы управления на такой панели инструментов перерисовываются в соответствии с цветами новой палитры.

Обратите внимание, что в компоненте `TToolBar`, перенесенном из Палитры инструментов на форму вручную, это свойство отсутствует.

Все компоненты настройки цветовой палитры имеют один метод-обработчик

```
property OnColorChange: TnotifyEvent;
```

который вызывается при изменении любого цвета палитры.

Резюме

Элементы управления — поистине неисчерпаемая тема. Надеемся, что читатель воспримет данную главу как руководство к последующему углубленному анализу того или иного элемента в свете стоящих перед ним задач.

С элементами управления вам так или иначе придется сталкиваться во всех последующих главах.

В оформлении пользовательского интерфейса операционной системы Windows XP появилось новшество — визуальные стили, которые позволяют настраивать не только внешний вид элементов управления, но и их поведение и даже управлять отрисовкой отдельных частей элементов управления. Это стало возможным благодаря системной библиотеке ComCtl32.dll шестой версии.

Глава 9

Меню и действия

Меню, как элемент программного интерфейса, присутствовали уже в самых первых программах, работающих в персональных компьютерах. За прошедшие годы роль меню не уменьшилась — любое современное приложение имеет меню. Причем все это время меню совершенствовалось. Такая популярность этого элемента интерфейса приложений объясняется просто: меню обеспечивает стандартный и удобный доступ к функциям приложения и прекрасно структурирует их в однородные группы.

В VCL Delphi включены два компонента меню:

- `TMainMenu` — обеспечивает создание полосы главного меню приложения;
- `TPopupMenu` — представляет собой всплывающее меню, которое может быть придано любому визуальному компоненту.

Компоненты меню унаследованы от классов `TMenu` и `TComponent`. Поэтому, строго говоря, меню не являются элементами управления и становятся визуальными только при выполнении приложения. Во время разработки меню конструируются при помощи специального Редактора меню.

Главное меню формы частично работоспособно уже во время разработки приложения.

В Delphi у разработчика есть и альтернативная возможность создания пользовательского интерфейса с меню на базе новых компонентов `TActionManager`, `TActionMainMenuBar`, `TActionToolBar` и примкнувшего к ним `TCustomizeDlg`.

Но в любом случае программисту пригодятся действия — объекты специализированного класса `TAction`, облегчающие работу по разработке пользовательского интерфейса приложения.

В этой главе рассматриваются следующие вопросы.

- Программная реализация меню.
- Главное меню.
- Всплывающее меню.

- Действия.
- Списки действий.
- Менеджер действий.

Редактор меню

Для работы с компонентами меню во время разработки используется специализированный Редактор меню (рис. 9.1). Он открывается при щелчке на кнопке свойства `Items` компонента меню в Инспекторе объектов или при двойном щелчке на компоненте и обеспечивает отображение всех элементов меню, представленных в свойстве `Items`.

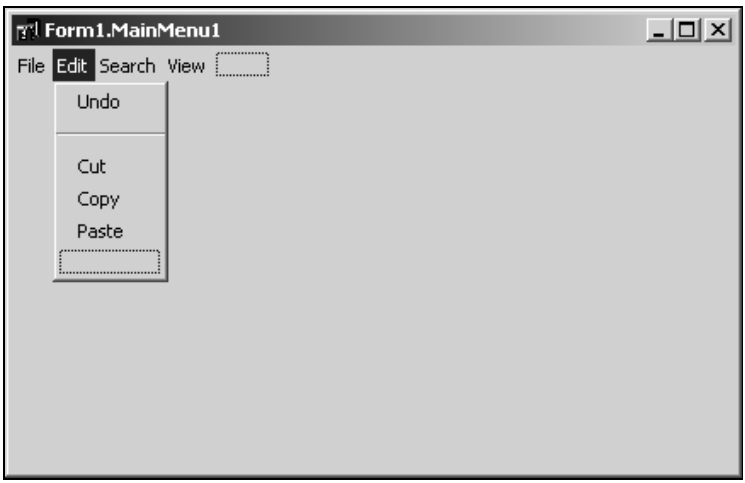


Рис. 9.1. Редактор меню

Редактор меню активно взаимодействует с Инспектором объектов. Для каждого выбранного в Редакторе элемента в Инспекторе объектов появляются значения его свойств. Перемещаясь по элементам, разработчик может редактировать их свойства.

Для того чтобы добавить новый элемент в конец меню, достаточно выбрать пустой элемент, который всегда имеется в конце списка, и заполнить его свойства.

Для вставки и удаления элементов предусмотрены команды всплывающего меню Редактора или соответствующие клавиши.

Для выбранного элемента меню в Инспекторе объектов активизируется свойство `Caption`, что позволяет быстро вводить текст команды меню и пе-

реходить к следующему элементу. Если в тексте команды встречается символ амперсанта `&`, то следующий за ним символ автоматически становится горячей клавишей. Например, свойство

```
Caption := '&Close'
```

в меню представляет команду **Close**.

Если в качестве текста команды введен символ -, то этот элемент меню отображается как разделитель.

Для того чтобы сделать из обычной команды вложенное меню достаточно на активном элементе нажать комбинацию клавиш <Ctrl>+<→> или выбрать команду **Create Submenu** из всплывающего меню. В результате в правой части элемента появляется знак вложенного меню, для которого можно создавать команды обычным порядком.

Как работает меню

По существу, программная реализация любого меню должна быть очень проста. С каждой командой меню должен быть связан метод-обработчик, содержащий программный код для вызова необходимых функций приложения.

Основные возможности, обеспечивающие функционирование меню, реализованы в базовом классе меню `TMenu`.

В основе класса меню лежит свойство, представляющее собой коллекцию элементов меню. Каждый элемент представляет собой отдельную команду меню, которая может быть конечной (вызывать функцию приложения) или раскрывать меню более низкого уровня.

```
property Items: TMenuItem; default;
```

К отдельному элементу меню можно обратиться следующим образом:

```
SomeMenu.Items[1].Caption := 'Edit';
```

Каждому элементу меню можно придать собственное изображение. Для этого есть свойство

```
property Images: TImageList;
```

Картинки должны содержаться в компоненте `TImageList`.

Класс `TMenuItem` объединяет элементы меню, его свойства и методы представлены в табл. 9.1. Обратите внимание, что здесь также имеется свойство `Items` типа `TMeuItem`, которое необходимо в случае, если данный элемент является меню следующего уровня. Такая вложенность может реализовываться сколь угодно глубоко.

Таблица 9.1. *Основные свойства и методы класса* `TMenuItem`

Объявление	Описание
Свойства	
`property Action: TBasicAction;`	Определяет действие, связанное с элементом меню
`property Bitmap: TBitmap;`	Содержит изображение, отображающееся вместе с элементом меню
`type TMenuBreak = (mbNone, mbBreak, mbBarBreak);` `property Break: TMenuBreak;`	Управляет представлением элементов в панели меню. `mbNone` — элементы следуют друг за другом. `mbBreak` — начиная с этого элемент, команды располагаются в соседнем столбце. `mbBarBreak` — начиная с этого элемента команды располагаются в соседнем столбце. Между столбцами расположен разделитель
`property Caption: string;`	Текст команды меню
`property Checked: Boolean;`	Управляет флажком в элементе меню
`property Default: Boolean;`	Управляет выделением элемента меню
`property Enabled: Boolean;`	Управляет доступностью элемента меню
`property GroupIndex: Byte;`	Используется при слиянии двух меню
`property Items[Index: Integer]: TMenuItem; default;`	Список элементов меню
`property MenuIndex: Integer;`	Содержит индекс элемента в родительском меню
`property RadioItem: Boolean;`	Управляет включением переключателя около элемента меню
`property ShortCut: TShortCut;`	Содержит объект горячей клавиши элемента меню
`property Visible: Boolean;`	Управляет видимостью элемента меню

Таблица 9.1 *(окончание)*

Объявление	Описание
Методы	
`procedure Add(Item: TMenuItem);`	Добавляет в конец меню новый элемент
`procedure Click; virtual;`	Генерирует выбор элемента меню
`procedure Delete(Index: Integer);`	Удаляет элемент меню
`function HasParent: Boolean; override;`	Определяет наличие родительского меню
`function IndexOf(Item: TMenuItem): Integer;`	Возвращает индекс элемента меню
`procedure Insert(Index: Integer; Item: TMenuItem);`	Вставляет новый элемент меню перед текущим
`procedure Remove(Item: TMenuItem);`	Удаляет элемент меню
Методы-обработчики событий	
`property OnClick: TNotifyEvent;`	Вызывается при щелчке на элементе
`type TMenuDrawItemEvent = procedure (Sender: TObject; ACanvas: TCanvas; ARect: TRect; Selected: Boolean) of object;` `property OnDrawItem: TMenuDrawItemEvent;`	Вызывается при необходимости перерисовки элемента
`type TMenuMeasureItemEvent = procedure (Sender: TObject; ACanvas: TCanvas; var Width, Height: Integer) of object;` `property OnMeasureItem: TMenuMeasureItemEvent;`	Вызывается для определения размеров элемента при его перерисовке

Большинство свойств класса `TMenuItem` обеспечивают изменение атрибутов элемента меню (видимость, выделение, дополнительные элементы управления).

Свойство `Break` изменяет внешний вид панели меню, содержащей данный элемент. На рис 9.2 показан результат использования свойства `Break` (соответственно значения `mbBreak` и `mbBarBreak` присвоены элементу **Экран**).

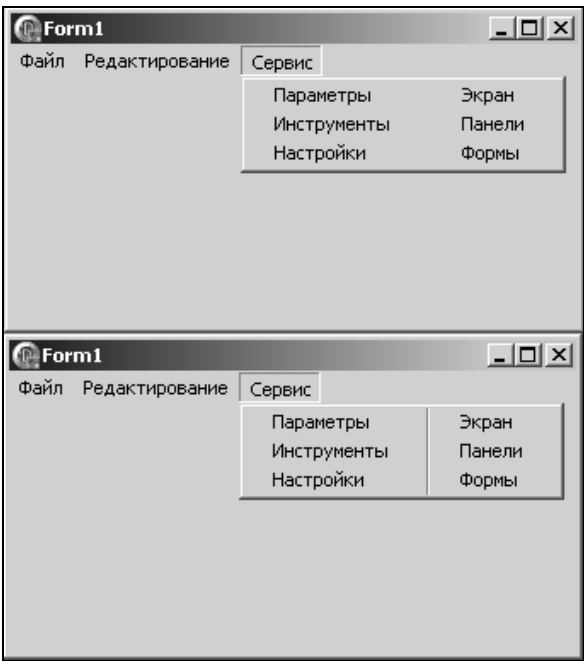


Рис. 9.2. Виды представления команд меню

Набор методов обеспечивает управление элементами программными средствами (вместо Редактора меню), в том числе их создание и удаление.

Метод-обработчик `OnClick` обеспечивает передачу выбора команды меню в приложение. В нем разработчик должен предусмотреть реакцию приложения на это событие. Часто команда осуществляет вызов новой формы:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
 SomeForm.ShowModal;
end;
```

Для того чтобы связать меню с конкретным элементом управления, необходимо задействовать свойства этого элемента управления. Свойство `Menu` в классе формы `TForm` связывает с формой компонент главного меню. Свойство `PopupMenu` в других визуальных компонентах обеспечивает реализацию компонента всплывающего меню.

Главное меню приложения

Компонент `TMainMenu` инкапсулирует главное меню. Он происходит от класса `TMenu`, а для создания элементов меню использует Редактор меню.

Для того чтобы связать форму и главное меню, применяется свойство `Menu` класса `TForm`. В нем из списка доступных в форме компонентов `TMainMenu` выбирается нужный компонент. При размещении на форме первого компонента `TMainMenu` он связывается с формой автоматически.

Из собственных свойств и методов компонента необходимо отметить свойства и методы, обеспечивающие слияние двух главных меню при наличии в приложении двух форм с собственными меню. Например, дочерняя форма может добавлять собственные команды к меню главного приложения.

Для автоматического слияния двух меню предусмотрено свойство

```
property AutoMerge: Boolean;
```

Меню дочерней формы должно включаться в состав меню главной формы. Для этого свойство `AutoMerge` дочерней формы должно иметь значение `True`. Это же свойство для меню главной формы должно иметь значение `False`.

Для этих же целей существует метод

```
procedure Merge(Menu: TMainMenu);
```

Он применяется в случаях, когда слияние меню желательно только при возникновении определенной ситуации и не должно осуществляться автоматически. Причем метод вызывается компонентом меню главного окна, а в качестве параметра передается указатель на компонент меню дочернего окна.

Непосредственно на результат слияния влияет свойство

```
property GroupIndex: Byte;
```

которое имеется у каждого элемента меню. Оно позволяет вставлять, заменять, добавлять в начало и конец меню главного окна новые элементы.

По умолчанию все элементы меню имеют одно значение свойства `GroupIndex`. Обычно это 0. Значение свойства элемента меню дочернего окна сравнивается со значениями свойства элементов меню главного окна. На этой основе осуществляется формирование объединенного меню. Его элементы ранжируются в соответствии с возрастанием значений свойства `GroupIndex`.

Если значения свойства в элементах объединяемых меню совпадают, то элемент главного меню заменяется на элемент дочернего. Если при этом заменяется несколько элементов одновременно, то замена осуществляется в соответствии с их взаимным положением. Первый элемент главного меню с

тем же значением свойства `GroupIndex` заменяется на первый по порядку элемент дочернего меню, второй элемент меняется на второй по порядку элемент и т. д.

Если значения свойства `GroupIndex` элементов дочернего меню лежат между значениями этого свойства элементов главного, то данные элементы дочернего меню вставляются между элементами главного.

Если значение свойства элемента дочернего меню больше максимального значения свойства элементов главного, то элементы дочернего меню добавляются в конец главного.

Если значение свойства элемента дочернего меню меньше минимального значения свойства элементов главного, то элементы дочернего меню добавляются в начало главного.

В качестве небольшого примера рассмотрим приложение DemoMerge (рис. 9.3).

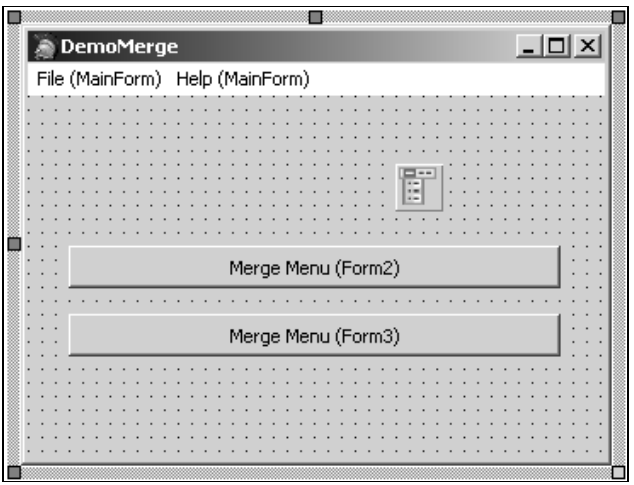


Рис. 9.3. Главная форма проекта DemoMerge

При щелчке на кнопке **Merge Menu (Form2)** появляется дочерняя форма `Form2`. Ее меню объединяется с меню главной формы за счет свойства `AutoMerge`, при этом аналогичное свойство главной формы имеет значение `False`. При щелчке на кнопке **Merge Menu (Form3)** при помощи метода `Merge` меню формы `Form3` к главному меню добавляются команды меню формы `Form3`. При этом сама форма `Form3` не открывается. Листинг 9.1 содержит код секции `implementation` модуля главной формы проекта DemoMerge, а листинг 9.2 — модуля формы `Form3` проекта DemoMerge.

Листинг 9.1. Секция `implementation` модуля главной формы проекта DemoMerge

```
implementation
{$R *.dfm}
uses Unit2, Unit3;
procedure TMainForm.BitBtn1Click(Sender: TObject);
begin
  Form2.Show;
end;
procedure TMainForm.BitBtn2Click(Sender: TObject);
begin
  MainMenu.Merge(Form3.Form3Menu);
end;
end.
```

Значения свойства `GroupIndex` для главного и дочерних окон можно просмотреть при помощи Редактора меню.

Благодаря механизму слияния в меню главного окна внедряются команды меню других форм.

Примечание

Команды меню главного окна не имеют связанных методов-обработчиков и предназначены для иллюстрации работы свойства `GroupIndex`.

Обратите внимание, что при добавлении в главное меню элементов меню еще одного дочернего окна механизм слияния не учитывает наличие в главном меню уже добавленных ранее элементов другого дочернего окна.

Листинг 9.2. Секция `implementation` модуля формы `Form3` проекта DemoMerge

```
implementation
{$R *.dfm}
uses Unit1;
procedure TForm3.CloseForm31Click(Sender: TObject);
begin
  MainForm.MainMenu.UnMerge(Form3Menu);
  Close;
end;
end.
```

При использовании для слияния меню метода `Merge` необходимо предусмотреть восстановление исходного состояния главного меню при закрытии дочерней формы (см. листинг 9.2) при помощи метода `UnMegre`:

```
procedure Unmerge(Menu: TMainMenu);
```

Для этого из формы `Form3` в главное меню добавляется команда, выполняющая это действие.

Всплывающее меню

Компонент `TPopupMenu` инкапсулирует всплывающее меню. Его настройка также осуществляется с помощью Редактора меню. Для того чтобы связать всплывающее меню с элементом управления, применяется свойство `PopupMenu`. В нем из списка доступных компонентов всплывающего меню выбирается необходимый.

Свойство

```
property AutoPopup: Boolean;
```

обеспечивает разворачивание меню при щелчке правой кнопкой мыши на элементе управления.

Свойство

```
property PopupComponent: TComponent;
```

содержит указатель на компонент, использующий данное всплывающее меню.

Метод

```
procedure Popup(X, Y: Integer); virtual;
```

разворачивает всплывающее меню в точке экрана с координатами `X` и `Y`.

При открытии всплывающего меню вызывается метод-обработчик

```
property OnPopup: TNotifyEvent;
```

Действия. Компонент *TActionList*

Пользовательский интерфейс современных приложений весьма многообразен, и зачастую один и тот же результат можно получить разными способами: щелчком на кнопке на панели инструментов, выбором пункта меню, нажатием комбинации клавиш и т. п. Можно решить проблему "в лоб" и повторить один и тот же код два, три раза и т. д. Недостатки такого подхода очевидны, не обязательно их комментировать. Можно воспользоваться Инспектором объектов и назначить пункту меню тот же обработчик события, что и кнопке, благо событие `OnClick` имеет везде одинаковый синтак-

сис. Этот способ неплох, но при большом количестве взаимных ссылок легко запутаться.

Наконец, современный способ — это воспользоваться компонентом `TActionList` и разом решить эти проблемы. Как следует из названия, этот компонент является централизованным хранилищем, где воздействия со стороны пользователя связываются с реакциями на них.

Действием (Action) будем называть операцию, которую пользователь хочет произвести, воздействуя на элементы интерфейса. Тот компонент, на который он хочет воздействовать, называется *целью действия (Action Target)*. Компонент, посредством которого действие инициировано (кнопка, пункт меню), называется *клиентом действия (Action client)*. Таким образом, в иерархии классов Delphi действие `TAction` — это невизуальный компонент, который играет роль "черного ящика", получающего сигнал от одного или нескольких клиентов и выполняющих действия над одной (или разными) целями.

Примечание

Действия могут работать, только будучи объединены в список компонентов `TActionList` или `TActionManager`. Вне этих компонентов применение действий невозможно.

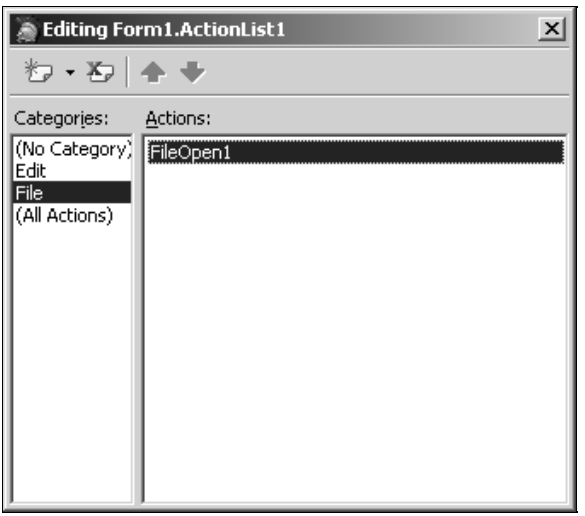


Рис. 9.4. Внешний вид редактора действий компонента `TActionList`

Спроектировав на бумаге пользовательский интерфейс, начните работу с помещения на форму компонента `TActionList`. Он находится в Палитре

инструментов в категории **Standard**. После этого следует запустить редактор списка действий двойным щелчком мышью на компоненте или с помощью контекстного меню (рис. 9.4).

Теперь можно начинать добавление действий. Для программиста предусмотрен набор типовых, наиболее часто встречающихся действий, которые описаны в следующем разделе.

Помимо них можно вставить и обычное действие, которое получит имя `Action1`. Итак, что же собой представляет действие?

События, связанные с действиями

Компонент `TAction` реагирует на три события: `OnExecute`, `OnUpdate` и `OnHint`. Первое — и самое главное — должно быть как раз реакцией на данное действие. Это событие возникает в момент нажатия кнопки, пункта меню — короче, при поступлении сигнала от клиента действия. Здесь, как правило, и пишется обработчик. Почему "как правило"? Потому что схема обработки сигнала четырехэтапная.

1. Сначала вызывается обработчик события `OnExecute` списка действий `TActionList`:

   ```
   property OnExecute: TActionEvent;
   TActionEvent = procedure (Action: TBasicAction; var Handled: Boolean)
   of object;
   ```

   Если обработчик этого события вами не предусмотрен или в параметре `Handled` он вернул значение `False`, происходит генерация следующего события — шаг 2.

2. Вызывается обработчик события `OnActionExecute` глобального объекта `Application` (тип события тот же — `TActionEvent`). Если и оно не обработало сигнал действия, переходим к следующему шагу.

3. Вызывается обработчик события `OnExecute` самого действия (объекта типа `TAction` или его потомка).

4. Если первые три шага не обработали ситуацию (вернули `False`), то, вероятно, это было связано с неправильной целью (Target) действия. В качестве "последнего шанса" приложению посылается сообщение `CM_ACTIONEXECUTE`. В этом случае происходит поиск другой цели для данного действия (об алгоритме поиска цели см. далее).

Первые две возможности в этом списке встречаются относительно редко. Тем не менее они полезны, если вам нужно глобально добавлять, удалять, разрешать, запрещать действия.

Введение события `OnUpdate` является очень хорошей находкой, о нем напишем подробно. И автор этих строк, и, возможно, вы, потратили немало

времени, чтобы в разрабатываемых программах элементы управления находились в актуальном состоянии. Если, скажем, вашей программой открыт первый файл, то нужно активировать ряд кнопок и пунктов меню (**Save**, **Save as**, **Print** и т. п.); как только закрыт последний — отключить их. Если в буфере обмена есть что-то подходящее, нужно активировать пункт меню и кнопку **Paste**, если нет — отключить. В результате код, отслеживающий это, у неопытных программистов размазывается по всему приложению. А ведь можно поступить проще. Событие `TAction.OnUpdate` возникает в моменты простоя приложения, т. е. тогда, когда оно не занято обработкой сообщений (цикл содержится в методе `Idle` объекта `Application`). Это гарантирует, что оно возникнет *до того*, как пользователь щелкнет мышью и увидит выпадающие пункты меню, поэтому можно успеть обновить их состояние. Пример использования события `OnUpdate`:

```
procedure TForm1.PasteActionUpdate(Sender: TObject);
begin
   TAction(Sender).Checked := Clipboard.HasFormat(CF_TEXT);
end;
```

Примечание

Перед вызовом `OnUpdate` также происходит четырехэтапная последовательность действий, точно такая же, как при `OnExecute`.

Третье событие имеет такой тип:

```
THintEvent = procedure (var HintStr: string; var CanShow: Boolean) of
object;
```

Оно вызывается тогда, когда от элемента управления требуется подсказка, связанная с данным действием. В обработчике события можно указать, будет ли что-нибудь показываться (параметр `CanShow`), и если да, то что именно (параметр `HintStr`).

Это были события, относящиеся к `TAction`. Сам компонент `TActionList` также имеет три события: `OnExecute`, `OnUpdate` и `OnChange`. О первых двух мы уже сказали, третье происходит в момент изменения списка (добавления или удаления действий).

Свойства, распространяемые на клиентов действия

Если у нескольких кнопок или пунктов меню общий обработчик, разумно потребовать, чтобы у них были и другие общие свойства. Так оно и реализовано в Delphi. В табл. 9.2 перечислены свойства, чье значение автоматически распространяется на всех клиентов данного действия.

Таблица 9.2. *Свойства компонента* `TAction`*, автоматически распространяемые на всех его клиентов*

Свойство	Назначение
`property Caption: string;`	Заголовок, связанный с действием
`property Hint: string;`	Подсказка к действию
`property Enabled: Boolean;`	Устанавливает, разрешено/запрещено ли действие
`property Checked: Boolean;`	Устанавливает, отмечено ли действие
`property GroupIndex: Integer;`	Индекс группы, в которую входит действие. Объекты `TAction` с одним значением этого свойства (причем большим нуля) имеют зависимое переключение. Если свойство `Checked` любого объекта из группы устанавливается в `True`, у остальных оно сбрасывается в `False`
`property AutoCheck: Boolean;`	Установка в `True` автоматически меняет значение свойства `Checked` на противоположное после каждого выполнения действия
`property ImageIndex: Integer;`	Индекс картинки в общем наборе картинок (набор указывается в свойствах родительского `TActionList`)
`property HelpType: THelpType;`	Указывает на тип значения, связывающего действие с разделом системы помощи (htKeyword/htContext)
`property HelpContext: THelpContext;`	Если свойство `HelpType` установлено в `htContext`, это свойство содержит ID раздела системы помощи
`property HelpKeyword: string;`	Если свойство `HelpType` установлено в `htKeyword`, это свойство содержит ключевое слово (термин), по которому происходит открытие соответствующего раздела системы помощи

Вы привыкли к программам с картинками в меню и на панелях инструментов? Действие также можно снабдить картинкой. Компонент `TActionList` связывается со списком картинок `TImageList`, а действие `TAction` — с конкретной картинкой через свойство `ImageIndex`. Таким образом, все элементы управления, связанные с действием (кнопки и пункты меню), будут иметь одну и ту же картинку, как показано на рис. 9.5. Впрочем, это относится ко всем остальным свойствам из табл. 9.2.

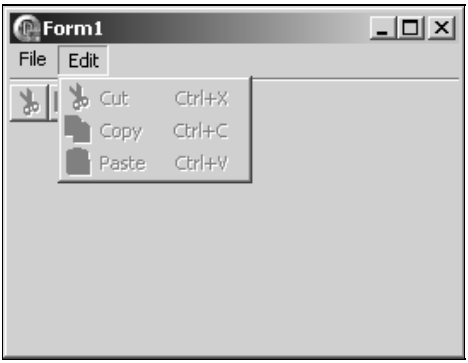


Рис. 9.5. Меню и панель инструментов используют один список действий

Прочие свойства

Чтобы связать с действием комбинацию горячих клавиш, выберите одну из возможных комбинаций в редакторе свойства `ShortCut`. Более того, в Delphi существует возможность добавлять не одну, а множество комбинаций горячих клавиш! Вторая и последующие должны быть заданы в свойстве `SecondaryShortCuts`. Когда вызывается редактор этого свойства, пользователь видит обычный редактор набора строк. И вводить комбинации нужно по принципу "как слышится, так и пишется": например, <Ctrl>+<F12>, <Shift>+<Alt>+<0>, и т. п., по одной комбинации на каждой строке.

Для упорядочивания все действия разбиты на категории:

```
property Category: string;
```

Это свойство содержит условное название категории, к которой относится действие, например, **File**, **Edit**, **Help** и т. п. Роль категории сводится к тому, чтобы объединить сходные действия при показе в `ActionList` или `ActionManager`. Названия категорий вы видите в панели меню, на самом верхнем уровне.

Иногда программисту все-таки необходимо знать, какой конкретно клиент (меню, кнопка) привел к выполнению действия. Узнать это можно, воспользовавшись значением свойства компонента `TAction`

```
property ActionComponent: TComponent;
```

перед вызовом `OnExecute` это свойство содержит указатель на клиента, инициировавшего действие. После вызова значение свойства очищается.

После того как определены действия и написан код, реагирующий на них, осталось поставить завершающую точку и связать их с пользовательским

интерфейсом. У большого числа элементов управления (например, всех кнопок и пунктов меню) есть опубликованное свойство `Action`. Если действия к этому моменту описаны, то в Панели инструментов из выпадающего списка среди возможных значений этого свойства достаточно выбрать нужное действие.

Стандартные действия

Те, кто собирается пропустить этот раздел, считая, что в нем описаны очевидные вещи, сильно ошибаются! Как раз в применении стандартных действий разработчики Borland продвинулись очень сильно. Кто хочет в этом убедиться, может открыть пример WordPad, поставляемый с Delphi 2005 (рис. 9.6). Этот полнофункциональный текстовый редактор, построенный полностью на стандартных действиях, содержит всего две строчки кода.

Шаблоны и заготовки для типовых меню и кнопок появились еще в самой первой версии Delphi. Но сейчас действия по-настоящему стали действиями. Это значит, что раньше заготовка содержала только подходящий заголовок. Теперь они содержат в себе все субкомпоненты, весь программный код и делают всю необходимую работу сами.

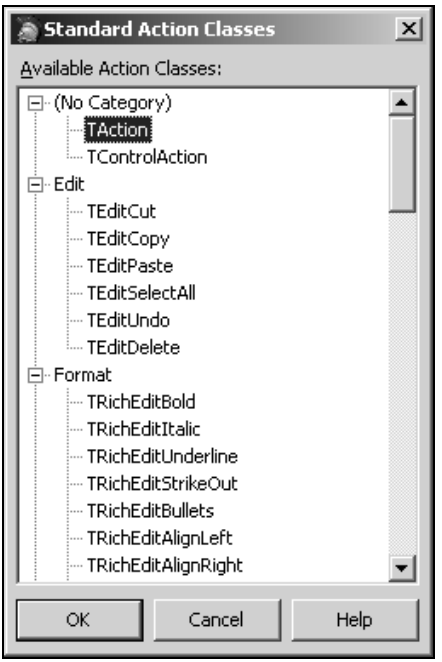


Рис. 9.6. Окно выбора стандартных действий

Возьмем, например, действие `TFileOpen`. Оно уже содержит внутри компонент типа `TOpenDialog`, показывающий список открываемых файлов. Вместо ручного программирования процедуры задания имени файла теперь нужно написать обработчик события `TFileOpen.OnAccept` (если пользователь ввел в диалоге **Ok**) или `OnCancel` (если отказался от открытия файла). Вот так выглядит весь программный код приложения WordPad:

```
procedure TForm1.FileOpen1Accept(Sender: TObject);
begin
  RichEdit1.Lines.LoadFromFile(FileOpen1.Dialog.FileName);
end;
procedure TForm1.FileSaveAs1Accept(Sender: TObject);
begin
  RichEdit1.Lines.SaveToFile(FileSaveAs1.Dialog.FileName);
end;
```

С точки зрения программирования, стандартное действие — это класс-потомок `TCustomAction`. Классы действий описаны в трех модулях: более распространенные в `StdActns`, менее — в `ExtActns`, действия с наборами данных содержатся в `DBActns`. Наконец, два действия, работающие со списками, — `TStaticListAction` и `TVirtualLitAction` — описаны в отдельном модуле `ListActns`.

Для выполнения ряда стандартных действий нужно определить понятие "цели" действия (Action Target). Под целью понимается компонент, в отношении которого будет совершаться данное действие. Так, операции редактирования могут выполняться, когда на форме активен текстовый элемент управления (`TEdit`, `TMemo` и т. п.). У любого действия (потомка `TBasicAction`) есть три метода:

- `function HandlesTarget(Target: TObject): Boolean; virtual;`
- `procedure UpdateTarget(Target: TObject); virtual;`
- `procedure ExecuteTarget(Target: TObject); virtual;`

Метод `HandlesTarget` проверяет, применимо ли действие к цели `Target`. Если да, то действие производится вызовом метода `ExecuteTarget`. Если нет, поиск подходящей цели продолжается.

Цель определяется по следующему алгоритму.

1. Первым кандидатом является активный элемент управления на форме (свойство `ActiveControl`).
2. Когда такового нет или он не является подходящим (метод `HandlesTarget` вернул `False`), целью становится текущая форма, получившая сигнал о действии.

3. Если и она не подходит, происходит рекурсивный перебор всех компонентов на форме в поисках первого подходящего.

В ряде случаев вы можете произвести действие над желаемым компонентом, вызвав метод `ExecuteTarget` и передав в него в качестве параметра этот компонент.

Примечание

Стандартные действия редактирования (чьи имена начинаются с `TEdit`...) и поиска (`TSearch`...) применимы только к потомкам компонента `TCustomEdit`. Стандартные действия расширенного редактирования (имена начинаются с `TRichEdit`...) применимы только к потомкам `TCustomRichEdit`. Оконные стандартные действия (упорядочивание, смена, закрытие дочерних окон; имена начинаются с `TWindow`...) применимы только к формам многодокументного интерфейса, чье свойство `FormStyle` установлено в `fsMDIForm`.

Многие классы стандартных действий не требуют элемента управления — цели. Так устроены все действия, вызывающие стандартные диалоговые окна (выбор файла, цвета, шрифта, настройка принтера и т. п.). Чтобы отреагировать на такое действие, нужно написать обработчики следующих событий:

```
property BeforeExecute: TNotifyEvent;
property OnAccept: TNotifyEvent;
property OnCancel: TNotifyEvent;
```

Первое возникает до показа диалога, второе — после нажатия пользователем кнопки **Ok**, третье — после нажатия **Cancel**.

Примечание

Поскольку диалоги входят в действия в качестве дочерних компонентов, вы можете реагировать и на все "дочерние" события, которые происходят в соответствующем диалоге (`OnShow`, `OnCanClose`, `OnClose` и т. п.).

Поместив на форму стандартные действия, вы заметите, что все они имеют предопределенное значение свойства `ImageIndex`. Если так, то где то изображение, на которое эти индексы указывают? Вы можете раздобыть его, открыв демонстрационный проект WordPad (папка Demos\ActionBands в поставке Delphi 2005), откройте редактор компонента `ImageList1` и экспортируйте весь список в виде одного большого файла формата BMP.

Категория *Edit*

В эту категорию входят компоненты, которые работают с редактируемыми элементами-потомками `TCustomEdit`. Это, к примеру, `TEdit`, `TMemo`, `TMaskedEdit`,

TRichEdit, TLabeledEdit и др. Причем целью может являться не любой редактируемый элемент, а только тот, что имеет фокус ввода. К категории относятся: TEditCut, TEditCopy, TEditPaste, TEditSelectAll, TEditDelete и TEditUndo.

Категория *Search*

Действия поиска и замены тоже выполняются только над потомками TCustomEdit. И это не прихоть разработчиков Borland, а сделано для вашего же удобства. Раньше для поиска приходилось самому программировать события OnFind и OnReplace соответствующих диалогов, а сейчас необходимый код уже содержится внутри действий.

К компонентам этой категории относятся TSearchFind, TSearchFindFirst, TSearchFindNext и TSearchReplace.

Категория *Help*

С помощью этих действий вы можете вызвать справочную систему вашего приложения. Стандартные действия категории Help приведены в табл. 9.3.

Таблица 9.3. *Стандартные действия категории* `Help`

Компонент	Назначение
THelpContents	Показывает оглавление системы справки
THelpOnHelp	Показывает подсказку по использованию системы справки
THelpContext	Показывает справку по контексту активного элемента управления (причем он должен быть ненулевым)
THelpTopicSearch	Показывает окно поиска системы справки

Категория *File*

Эти действия, скорее всего, будут наиболее востребованы разработчиками. И они же являются довольно простыми в использовании. TFileOpen, TFileSaveAs, TFilePrintSetup — это оболочки над соответствующими диалогами. О том, как работать с такими действиями, описано ранее. Действие TFileExit вообще не требует комментариев, оно просто завершает приложение, закрывая его главную форму.

Особняком стоит только TFileRun.

Категория *Dialog*

Эта категория примыкает к предыдущей, в ней содержатся остальные пять типовых действий-диалогов: `TPrintDlg`, `TColorSelect`, `TFontEdit` (из модуля `StdActns`), `TOpenPicture`, `TSavePicture` (модуль `ExtActns`).

Категория *Window*

Эти действия стоит включать в интерфейс, только если вы используете многодокументный интерфейс (MDI). Названия компонентов говорят сами за себя: `TWindowClose`, `TWindowCascade`, `TWindowTileHorizontal`, `TWindowTileVertical`, `TWindowMinimizeAll`, `TWindowArrange`.

Категория *Tab*

Здесь всего два компонента: `TNextTab` и `TPreviousTab`. Если цель действия — набор страниц (`TPageControl`), они переключат его на следующую и предыдущую страницу соответственно.

Категория *List*

В этой категории выделяются две группы действий. Первые пять из них автоматизируют работу с выделенными элементами списков. Оставшиеся два — `TStaticListAction` и `TVirtualListAction` — требуют отдельного рассмотрения.

Действия работают с компонентом `TListBox`, а в среде Kylix — еще и с `TListView` (не путать с одноименным компонентом для Windows — он не годится для данной категории). Подходит также и `TComboBox`.

В отличие от многих других действий, члены этой категории могут явно связываться с нужным вам списком. Если задано значение свойства `ListControl`, то все действия производятся над ним. Если нет, то выбирается активный список из числа имеющихся на форме. Действия по работе с выделенными элементами списков приведены в табл. 9.4.

Таблица 9.4. *Действия по работе с выделенными элементами списков*

Действие	Назначение
`TListControlSelectAll`	Выделяет все элементы списка. Активно, только если у списка свойство `MultiSelect` установлено в `True`
`TListControlClearSelection`	Отменяет выделение элементов в списке
`TListControlDeleteSelection`	Удаляет выделенные элементы

Таблица 9.4 *(окончание)*

Действие	Назначение
`TListControlCopySelection`	Копирует выделенные элементы списка в список `Destination`
`TListControlMoveSelection`	Переносит выделенные элементы списка в список `Destination`

Особняком стоят два действия: `TStaticListAction` и `TVirtualListAction`.

Они, по замыслу разработчиков, являются централизованными хранилищами элементов для многих списков. Причем элементы списка могут храниться сразу с заданными картинками (т. е. свойствами `ImageIndex`) и указателями на сопутствующие данные.

Дальнейшее просто — разработчик выбирает нужные компоненты `TListBox`, `TComboBox` и т. п., и в их свойстве `Action` указывает на действие — хранилище. Опубликовано свойство `Action` у компонента `TComboBoxEx` (впервые появившегося в Delphi 6). У остальных потомков `TControl` это свойство относится к группе видимости `public`, поэтому вы можете сделать присвоение при запуске приложения (в методе-обработчике `OnCreate` главной формы).

Если действие и компонент-список связаны, то должны происходить две вещи.

1. При изменении текущего элемента в любом из компонентов происходит синхронное изменение во всех остальных.
2. Когда пользователь выбирает один из элементов списка, выполняется действие, связанное с этим списком, и вызывается метод-обработчик:

```
type TItemSelectedEvent = procedure(Sender: TCustomListAction;
Control: TControl) of object; property OnItemSelected:
TitemSelectedEvent;
```

Категория *Internet*

Здесь всего три — типовых для пользователя Сети — действия.

Действие `TBrowseURL` открывает URL, заданный в одноименном свойстве. Поскольку это происходит при помощи функции `ShellExecute`, для просмотра открывается браузер, зарегистрированный в системе по умолчанию.

Действие `TSendMail` запускает программу — почтового клиента для отправки письма (с помощью интерфейса MAPI). Текст письма вы можете задать в свойстве `Text`. Но! Ни получателя, ни тему, ни вложений задать нельзя, — все это придется делать вручную в почтовой программе. При желании полностью автоматизировать процесс отправки вам придется породить дочер-

ний компонент от TSendMail, где и перекрыть метод ExecuteTarget. Исходные тексты — в модуле ExtActns.

Наконец, самый сложный компонент — TDownloadURL. Он позволяет загрузить содержимое с адреса URL и сохранить его на локальной машине под именем FileName.

Поскольку загрузка — процесс долгий, в то время, пока она происходит, периодически возникает событие

```
property OnDownloadProgress: TDownloadProgressEvent;
TDownloadProgressEvent = procedure(Sender: TDownLoadURL; Progress,
    ProgressMax: Cardinal; StatusCode: TURLDownloadStatus; StatusText:
    String;
    var Cancel: Boolean) of object;
```

Перечислим параметры обработчика этого события.

- Progress и ProgressMax — текущее и максимальное значения показателя хода скачивания. Во-первых, не все HTTP-серверы правильно сообщают о размере ответа. Во-вторых, для некоторых типов файлов (например, HTML) эти параметры вычисляются не всегда верно (вы можете это видеть в Internet Explorer). В-третьих, из-за маршрутизации пакетов ожидать ритмичного изменения параметра Progress не следует. Поэтому пользователю надо показывать соотношение Progress/ProgressMax.

  Примечание

  Во многих случаях значение ProgressMax может быть равно нулю. В этом случае о ходе загрузки численно судить нельзя. Информацию несут другие параметры события.

- StatusCode и StatusText — код, описывающий текущее состояние операции и соответствующий ему текст. Список возможных кодов содержит около 30 значений. Для тех, кто знает протокол HTTP и хочет разобраться в этом глубже, следует обратиться к описанию интерфейса IBindStatusCallback в MSDN. Если же вам достаточно показать пользователю текст, то он содержится во втором параметре. По содержанию он представляет примерно то же, что вы видите при загрузке файлов с помощью Internet Explorer.
- Cancel — одинаков для всех долго продолжающихся операций. Установив его в True, вы можете прервать выполнение загрузки.

Категория *Format*

Эти действия представляют собой расширенные операции редактирования для "продвинутого" редактора TRichEdit. Эти операции должны быть зна-

комы вам по программе WordPad из состава Windows. В крайнем случае откройте демонстрационный пример Delphi с тем же названием — там присутствуют действия этой категории и подавляющее большинство остальных. В списке присутствуют `TRichEditBold`, `TRichEditItalic`, `TRichEditUnderline`, `TRichEditStrikeout` (установка стиля шрифта), `TRichEditBullets` (значки абзацев), `TRichEditAlignLeft`, `TRichEditAlignRight`, `TRichEditAlignCenter` (выравнивание текста).

Категория *Dataset*

Эти действия можно увидеть, например, в качестве кнопок на любом компоненте `TDBNavigator`: `TDataSetFirst`, `TDataSetPrior`, `TDataSetNext`, `TDataSetLast`, `TDataSetDelete`, `TDataSetInsert`, `TDataSetEdit`, `TDataSetPost`, `TDataSetCancel`, `TDataSetRefresh`. Читатель задаст вопрос: а как действие связывается с набором данных? Очень просто: через дополнительное (для данной категории) свойство `DataSource`. Если источник данных существует и связан с существующим набором данных (свойство `DataSource.DataSet`), то действие выполняется над ним.

Категория *Tools*

Здесь содержится один-единственный член: `TCustomizeActionBars`. Будучи вызванным, это действие вызывает диалог настройки панелей действий, относящихся к компоненту `TActionManager`, о котором, собственно, сейчас и пойдет речь.

Компонент *TActionManager*

Если вы не думаете о переносе своего приложения в среду Linux, то имеются все основания воспользоваться потомком `TActionList` — компонентом `TActionManager` (далее в тексте — менеджер действий). Более современный и "продвинутый", он обеспечит вас многими дополнительными возможностями. Итак, обо всем по порядку.

Будучи применен сам по себе, `TActionManager` ничем не отличается от предшественника. Отличия проявляются, если действия из этого компонента разместить на специальных панелях `TActionMainMenuBar` (будем называть его панелью главного меню) и `TActionToolBar` (далее — панель действий).

На первой странице редактора `TActionManager` (вызывается двойным щелчком или пунктом **Customize** контекстного меню, показан на рис. 9.7) как раз и содержится список всех панелей, связанных с данным менеджером действий. Вы можете добавить новый или убрать `TActionToolBar` нажатием кно-

пок **New** и **Delete**. С `TActionMainMenuBar` так, по понятным причинам, поступить нельзя — меню полагается иметь одно.

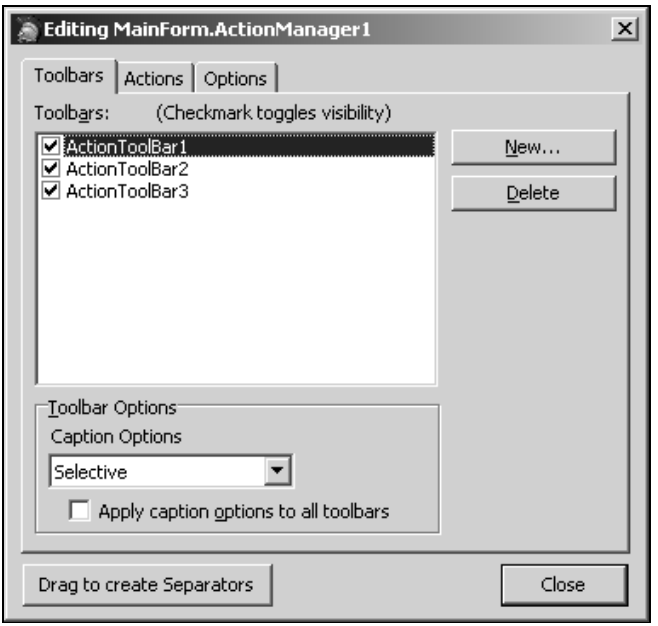


Рис. 9.7. Первая страница редактора свойств компонента `TActionManager`

Самый простой и рекомендованный Borland способ для связи действий с одной стороны и панелей меню и инструментов с другой — это перетаскивание (drag-and-drop). На второй странице редактора содержится список всех действий по категориям. И отдельное действие, и целую категорию можно брать и тащить мышкой на нужную панель (рис. 9.8).

Когда вы перетаскиваете действие на панель, на нем появляется специальный компонент, похожий на пункт меню или кнопку. Его роль — служить клиентом данного действия. Поэтому, естественно, он сразу и автоматически получает нужный `Caption`, `ImageIndex`, `Hint` и прочие общие для всех клиентов свойства, о которых говорилось ранее. Класс этого клиента — `TActionClientItem`; будем называть их псевдокнопками или псевдоэлементами.

При перетаскивании нет особых сложностей, но надо иметь в виду следующие аспекты:

- при перетаскивании всей категории на панель главного меню она появляется в виде пункта меню верхнего уровня и содержит при этом все свои дочерние действия;

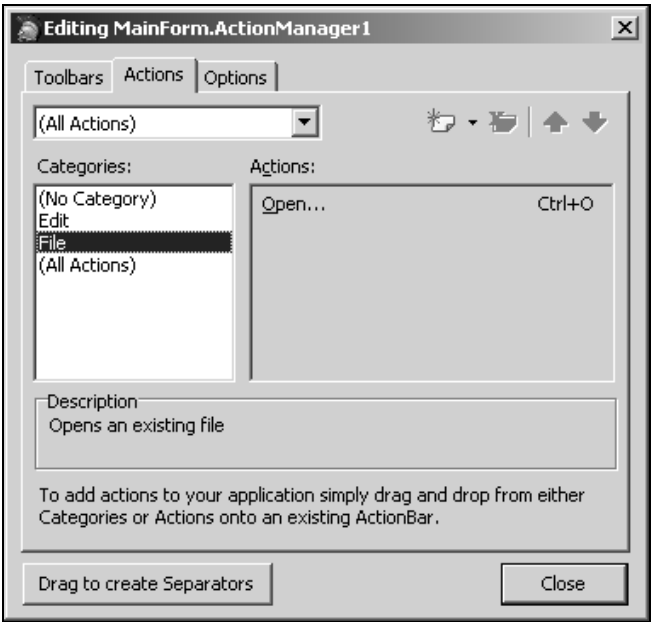


Рис. 9.8. Страница действий редактора свойств компонента `TActionManager`

- при перетаскивании всей категории на панель действий создаются псевдокнопки для всех дочерних действий в категории. Логично поступить по принципу: одна категория — одна панель действий, это будет полезно для настройки интерфейса пользователем;
- если вы ошиблись при перетаскивании, не нажимайте **Delete** — при этом удалится не только псевдокнопка, но и само действие. Перетяните ненужный псевдоэлемент за пределы панелей действий, тогда он будет удален;
- если вы уже перетянули категорию на панель главного меню, а потом решили добавить к ней действия, то вам придется убрать соответствующий ей псевдоэлемент и перетянуть всю категорию заново. Либо воспользоваться ручным способом, описываемым далее.

Изменение и настройка внешнего вида панелей

Теперь мы подошли к совсем новому свойству панелей `TActionMainMenuBar`. Теперь — как в Microsoft Office — возможно прятать редко используемые пункты меню. В самом деле, интерфейс программ подчас настолько сложен, что используют его на 100% минимальное количество пользователей. Поэтому элементы интерфейса, которые пользователь не задействовал в каком-то числе предыдущих запусков, автоматически прячутся.

Что и когда прятать, определяется свойством:

```
property PrioritySchedule: TStringList;
```

значение которого по умолчанию приведено в табл. 9.5. В левой колонке — общее число запусков приложения, в течение которых пользователь вызывал данное действие; в правой — число запусков, прошедших со времени последнего его использования. После истечения этого числа запусков клиенты действия упрятываются. Например, в меню они доступны не сразу, а после нажатия специального пункта с двумя стрелочками, обращенными вниз.

Естественно, чем чаще пользователь обращался к действию, тем дольше оно удержится на виду. Впрочем, если у вас другие взгляды на интерфейс, вы можете изменить значение `PrioritySchedule`.

Таблица 9.5. *Условия скрытия элементов панелей действий*

Число запусков приложения с обращением к действию	Число запусков приложения после последнего обращения
0, 1	3
2	6
3	9
4, 5	12
6—8	17
9—13	23
14—24	29
>= 25	31

Для подсчета величин, указанных в этой таблице, введены такие свойства:

- у объекта `TActionBars` (дочерний объект `TActionManager`) есть свойство

  ```
  property SessionCount: Integer;
  ```

  которое представляет собой глобальный счетчиков запусков приложения;
- а у каждого объекта `TActionClientItem` есть два свойства:
  - `property LastSession: Integer;` — в этом свойстве хранится номер последнего запуска приложения, в течение которого происходило обращение к данному элементу (нумерация совпадает с `SessionCount`);
  - `property UsageCount: Integer;` — счетчик использования элемента.

Но для того чтобы оперировать данными о количестве запусков, их надо где-то хранить. Организована система хранения следующим образом. У самого менеджера действий есть свойство

```
property FileName: TFileName;
```

которое указывает на файл, содержащий все настройки панелей, связанных с данным Менеджером. Он имеет формат двоичной формы и считывается/записывается при запуске и выходе из приложения. Впрочем, можно это сделать и в любой момент при помощи методов `LoadFormFile` и `SaveToFile`.

Все эти величины меняются автоматически, и их описание приведено для понимания сути происходящего. Сбросить же счетчик статистики запусков можно следующим образом: на этапе разработки на странице **Options** редактора свойств Менеджера действий есть кнопка **Reset Usage Count**. На этапе выполнения точно такая кнопка есть в диалоге `TCustomizeDlg`.

Помимо данных для подсчета запусков, в этом файле хранится и вся прочая информация о настройках. Последний из не упоминавшихся нами компонентов — диалог настройки `TCustomizeDlg`. Он представляет собой точную копию редактора свойств `TActionManager`, но позволяет делать все операции с действиями в режиме выполнения. Вызвать его просто — вызовом метода `Show`. А можно поступить еще проще. Есть стандартное действие `Customize` (категория `Tools`), которое и подразумевает вызов этого диалога.

Ручное редактирование коллекций панелей и действий

Перетаскивание имеет много достоинств, однако оно не всегда удобно. Поэтому предусмотрен и другой способ. Хоть он напрямую и не рекомендован в документации, но в ряде случаев более эффективен.

Рассмотрим работу с дочерними объектами менеджера действий, которые упакованы один в другой, как матрешки.

Итак, щелкнем мышью на `ActionManager` на форме и посмотрим на содержимое Инспектора объектов. Внутри него мы обнаружим сразу две "матрешки": свойство `ActionBars` содержит коллекцию ссылок на дочерние панели, а свойство `LinkedActionList` может содержать дополнительный список действий, отданных "в управление" данному менеджеру — например, для централизованной установки общих свойств.

Щелкнем на свойстве `ActionBars`. Появится редактор панелей (рис. 9.9), а в Инспекторе объектов обратим внимание на свойство объекта `ActionBars`

```
property Customizable: Boolean;
```

которое указывает, может ли коллекция редактироваться во время выполнения.

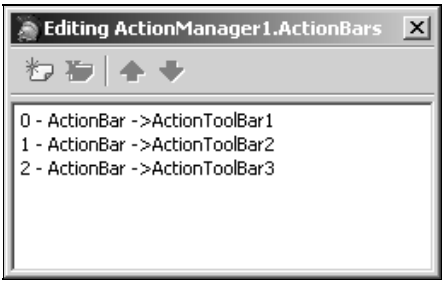


Рис. 9.9. Редактор коллекции панелей компонента `TActionManager`

В коллекции содержатся не сами панели, а их "заместители" — объекты типа `TActionBarItem`, которые на них указывают. Надпись на рисунке "1-ActionBar -> ActionToolBar1" говорит о том, что первый элемент коллекции связан с панелью `ActionToolBar1`. Вы можете добавлять и удалять элементы этой коллекции по мере необходимости, связывая их через свойство `ActionBar` с уже существующей панелью.

Через Инспектор объектов вы можете изменять внешний вид объектов типа `TActionBarItem` и соответствующих им панелей.

Свойство

```
property Color: TColor;
```

отвечает за фоновый цвет панели. Если вам изменения цвета недостаточно, в качестве фона выберите картинку:

```
property Background: TPicture;
```

которая будет расположена на панели в соответствии со значением свойства:

```
property BackgroundLayout: TBackgroundLayout;
TBackgroundLayout = (blNormal, blStretch, blTile, blLeftBanner,
blRightBanner);
```

Помимо внешнего вида, можно разрешить/запретить перетаскивание панелей и их дочерних элементов. Обратимся к свойству:

```
property ChangesAllowed: TChangesAllowedSet;
TChangesAllowed = (caModify, caMove, caDelete);
TChangesAllowedSet = set of TChangesAllowed;
```

Множество из трех возможных значений позволяет запретить те или иные нежелательные изменения для дочерних элементов панели. Если в него не включено `caDelete`, то элемент нельзя убирать (перетаскивать) с панели. Если нет `caMove`, нельзя передвигать внутри панели. Наконец, отсутствие

caModify означает запрет на изменение визуальных свойств (заголовка и т. п.).

Внутри TActionBarItem спрятаны еще две "матрешки": свойства Items и ContextItems. Оба свойства представляют собой коллекции объектов, указывающих на действия (класс коллекции TActionClients, класс элемента коллекции — TActionClientItem). Первое свойство указывает непосредственно на дочерние действия, второе — на действия, которые будут показаны в качестве всплывающего меню при нажатии правой кнопки мыши.

У коллекции TActionClients есть следующие заслуживающие особого упоминания свойства:

```
property CaptionOptions: TCaptionOptions;
TCaptionOptions = (coNone, coSelective, coAll);
```

Это свойство задает показ/отсутствие заголовков дочерних действий. В случае установки в coNone они не показываются, coAll — показываются все, coSelective — показываются в соответствии со значением ShowCaption дочернего объекта TActionClientItem. Это свойство можно также установить на первой странице редактора Менеджера действий в одноименном выпадающем списке.

```
property SmallIcons: Boolean;
```

указывает размер значков, соответствующих действиям. По умолчанию установлено в True (маленькие значки). Визуально оно доступно через тот же редактор — третья страница, флажок **Large Icons**.

```
property HideUnused: Boolean;
```

разрешает скрытие редко используемых действий, описанное в предыдущем разделе. Если вы не хотите пользоваться механизмом скрытия, на третьей странице редактора Менеджера действий и диалога TCustomizeDlg есть флажок **Menu show recent items first**. Сбросьте его, и свойства HideUnused у клиентов действий установятся в False.

И, наконец, коллекцию можно сделать нередактируемой. Для этого у нее есть свойство Customizable.

Ну вот, мы уже добрались до самой маленькой матрешки — TActionClientItem. Этот объект связывается напрямую с одним действием через свойство Action. Правда, в него можно спрятать еще меньшую матрешку — у него также есть свойства Items и ContextItems, которые необходимы при организации многоуровневых меню и меню, выпадающих из кнопок (точнее, псевдокнопок); напомним, что объекты TActionClientItem на панелях не являются ни кнопками, ни компонентами вообще.

Резюме

Хорошо знакомые со времен Delphi 1 составляющие интерфейса — меню (`TMainMenu`, `TPopupMenu`), кнопки (`TButton`, `TSpeedButton`), панели `TPanel` — постепенно уходят, уступая место компонентам с расширенной функциональностью. Центральным местом, где обрабатывается весь ввод пользователя, становится хранилище действий: `TActionList` или `TActionManager`.

В этой главе мы подробно рассмотрели оба компонента; читателю решать, на базе чего строить свой интерфейс.

С одной стороны, материал этой главы не оказывает прямого влияния на последующие. С другой, пользовательский интерфейс — та "печка", от которой пляшут при создании больших приложений. Поэтому имейте *главу 9* в виду при чтении всех прочих.

Глава 10

Списки и коллекции

Практически любое приложение должно уметь выполнять ряд стандартных операций по обработке каких-либо данных. К ним относятся загрузка данных при открытии приложения, представление данных в удобном виде внутри приложения, сохранение данных при завершении работы. Перечисленные действия необходимы и приложениям баз данных, и играм, и научным программам.

В принципе хранение и использование наборов значений можно обеспечить при помощи хорошо всем известных массивов. Однако их прямое применение требует от разработчика дополнительных усилий. Ведь при реализации программной логики необходимо добавлять в массив новые элементы, изменять существующие и удалять ненужные. Кроме этого, часто бывает необходимо найти элемент массива по значению. Все эти операции стандартны и повторяются для наборов любых типов данных.

Для решения перечисленных задач в Delphi предусмотрены специальные классы. Помимо хранения наборов значений в них реализованы свойства, позволяющие контролировать состояние списка и методы, обеспечивающие редактирование списка и поиск в нем отдельных элементов.

Для загрузки и сохранения данных существуют потоки — классы, инкапсулирующие механизмы доступа к различным хранилищам информации — файлам, памяти и т. д. Их общим предком является класс `TStream`.

Для работы со строковыми списками предназначены классы `TStrings` и `TStringList`.

Любые типы данных можно заносить в список указателей, который реализован в классе `TList`.

Использование наборов объектов (широко применяются в классах VCL), которые называются коллекциями, осуществляется при помощи классов `TCollection` и `TCollectionItem`.

В этой главе рассматриваются следующие вопросы.

- Что такое список. Как устроено основное свойство всех списков, объединяющее его элементы.

- Добавление, изменение и удаление элементов списка.
- Поиск заданного элемента.
- Механизм выделения памяти под элементы списка.
- Список строк.
- Список указателей.
- Чем отличается коллекция от списка.
- Коллекции.
- Использование потоков.

Список строк

Строковый тип данных часто встречается на практике. Во-первых, многие данные действительно необходимо представлять при помощи этого типа. Во-вторых, множество функций преобразования типов позволяют представлять числовые типы в виде строк, избегая тем самым проблем с несовместимостью типов.

По этой причине в первую очередь мы займемся изучением списка строк, который инкапсулирован в классах `TStrings` и `TStringList`. Первый класс является абстрактным и служит платформой для создания реально работающих потомков. Второй реализует вполне работоспособный список строк. Рассмотрим эти классы подробнее.

Класс *TStrings*

Класс `TStrings` является базовым классом, который обеспечивает потомков основными свойствами и методами, позволяющими создавать работоспособные списки строк. Его прямым предком является класс `TPersistent`.

Класс `TStrings` реализует все вспомогательные свойства и методы, которые обеспечивают управление списком. При этом методы, непосредственно добавляющие и удаляющие элементы списка, не реализованы и объявлены, как абстрактные.

Предупреждение

Попытка прямого вызова в приложении экземпляра класса `TStrings` вызовет ошибку использования абстрактного класса на этапе выполнения программы, а именно при попытке заполнить список значениями. Простая замена типа объектной переменной списка на `TStringList` делает приложение полностью работоспособным без какого-либо дополнительного изменения исходного кода.

Классы-наследники должны перекрывать методы добавления и удаления элементов списка. Реализованный в Delphi класс `TStringList` практически полностью повторяет функциональность предка, добавляя лишь несколько новых свойств и методов. Поэтому мы не станем останавливаться подробнее на классе `TStrings`, а перейдем сразу к его работоспособному потомку `TStringList`.

Класс *TStringList*

Класс `TStringList` обеспечивает реальную работу со списками строк в приложении. По существу, класс представляет собой оболочку вокруг динамического массива значений списка, представленного свойством `Strings`. Объявление свойства (унаследованное от `TStrings`) выглядит так:

```
property Strings[Index: Integer]: string read Get write Put; default;
```

Для работы со свойством имеются внутренние методы `Get` и `Put`, в которых есть внутренняя переменная `FList`:

```
type
 PStringItem = ^TStringItem;
   TStringItem = record
     FString: string;
     FObject: TObject;
   end;
   PStringItemList = ^TStringItemList;
   TStringItemList = array[0..MaxListSize] of TStringItem;

FList: PStringItemList;
```

Из ее объявления видно, что список строк представляет собой динамический массив записей `TStringItem`. Эта запись позволяет объединить саму строку и связанный с ней объект.

Максимальный размер списка ограничен константой

```
MaxListSize = Maxint div 16;
```

значение которой после нехитрых вычислений составит 134 217 727. Таким образом, видно, что строковый список Delphi теоретически конечен, хотя на практике гораздо чаще размер списка ограничивается размером доступной памяти.

Обращение к отдельному элементу списка может осуществляться через свойство `Strings` таким образом:

```
SomeStrings.Strings[i] := Edit1.Text;
```

или так:

```
SomeStrings[i] := Edit1.Text;
```

Оба способа равноценны.

При помощи простого присваивания можно задавать новые значения только тогда, когда элемент уже создан. Для добавления нового элемента служат методы `Add` и `AddStrings`.

Функция

```
function Add(const S: string): Integer;
```

добавляет в конец списка новый элемент, присваивая ему значение `S` и возвращая индекс нового элемента в списке.

Метод

```
procedure Append(const S: string);
```

просто вызывает функцию `Add`. Единственное отличие заключается в том, что метод не возвращает индекс нового элемента.

Метод

```
procedure AddStrings(Strings: TStrings);
```

добавляет к списку целый набор новых элементов, которые должны быть заданы другим списком, передаваемым в параметре `Strings`.

При необходимости можно добавить новый элемент в произвольное место списка. Для этого применяется метод

```
procedure Insert(Index: Integer; const S: string);
```

который вставляет элемент `S` на место элемента с индексом `Index`. При этом все нижеследующие элементы смещаются на одну позицию вниз.

Для удаления элемента списка служит метод

```
procedure Delete(Index: Integer);
```

Метод

```
procedure Move(CurIndex, NewIndex: Integer);
```

перемещает элемент, заданный индексом `CurIndex`, на новую позицию, заданную индексом `NewIndex`.

А метод

```
procedure Exchange(Index1, Index2: Integer);
```

меняет местами элементы с индексами `Index1` и `Index2`.

Довольно часто в списках размещается строковая информация следующего вида:

```
'Name=Value'
```

В качестве примера можно привести строки из файлов INI или системного реестра. Специально для таких случаев в списке предусмотрено представление строк в двух свойствах. В свойстве `Names` содержится текст до знака равенства. В свойстве `Values` содержится текст после знака равенства по умолчанию. Однако символ-разделитель можно заменить на любой другой с помощью свойства

```
property NameValueSeparator: Char;
```

Доступ к значениям свойства `Values` осуществляется по значению. Например, если в списке есть строка

```
City=Saint-Petersburg
```

то значение свойства `Value` будет равно

```
Value['City'] = 'Saint-Petersburg'
```

Кроме этого, значение свойства `Value` можно получить, если известен его индекс:

```
property ValueFromIndex[Index: Integer]: string;
```

Как видно из объявления внутреннего списка `FList`, с каждым элементом списка можно связать любой объект. Для этого есть свойство

```
property Objects[Index: Integer]: TObject;
```

Свойство `Strings` элемента и свойство `Objects` связанного с ним объекта имеют одинаковые индексы. Если строка не имеет связанного объекта, то свойство `Objects` равно `Nil`. Один объект может быть связан с несколькими строками списка одновременно.

Чаще всего объекты нужны для того, чтобы хранить для каждого элемента дополнительную информацию. Например, в списке городов для каждого элемента можно дополнительно хранить информацию о населении, площади, административном статусе и т. д. Для этого можно создать примерно такой класс:

```
TCityProps = class(TObject)
    Area: LongInt;
    Population: LongInt;
    Status: String;
  end;
```

Добавить объект к строке из списка можно методом `AddObject`:

```
function AddObject(const S: string; AObject: TObject): Integer; virtual;
```

Обратите внимание, что в параметре `AObject` необходимо передавать указатель на объект. Проще всего это сделать так:

```
SomeStrings.AddObject('SomeItem', TCityProps.Create);
```

Или же так

```
var SPb: TCityProps;
...
SPb := TCityProps.Create;  {Создание объекта}
SPb.Population := 5000000;
...
SomeStrings.Strings[i] := 'Санкт-Петербург';
SomeStrings.Objects[i] := SPb;     {Связывание объекта и строки}
...
```

Можно поступить и так (помните, что строка уже должна существовать):

```
...
SomeStrings.Strings[i] := 'Санкт-Петербург';
SomeStrings.Objects[i] := TCityProps.Create;
(SomeStrings.Objects[i] as TCityProps).Population := 5000000;
...
```

Аналогично методу `Insert`, элемент и связанный с ним объект можно вставить в произвольное место списка методом

```
procedure InsertObject(Index: Integer; const S: string; AObject: TObject);
```

При перемещении методом `Move` вместе с элементом переносится и указатель на связанный объект.

Обратите внимание на две особенности, связанные с удалением указателей на объекты и самих связанных объектов.

При удалении элемента списка удаляется только указатель на объект, а сам объект остается в памяти. Для его уничтожения необходимо предпринять дополнительные усилия:

```
...
for i := 0 to SomeList.Count – 1 do
 SomeList.Objects[i].Destroy;
...
```

Если при удалении связанного объекта необходимо выполнить некоторые действия, предусмотренные в деструкторе, приведение типов

```
TCityProps(SomeList.Objects[i]).Destroy;
```

выполнять не обязательно, поскольку нужный деструктор будет вызван автоматически, хотя в данном случае приведение типов ошибкой не является.

Метод

```
procedure Clear; override;
```

полностью очищает список, удаляя все его элементы.

Помимо перечисленных, класс `TStringList` обладает рядом дополнительных свойств и методов. Вспомогательные свойства класса обеспечивают разработчика информацией о состоянии списка. Дополнительные методы осуществляют поиск в списке и взаимодействие с файлами и потоками.

Свойство только для чтения

```
property Count: Integer;
```

возвращает число элементов списка.

Так как основу списка составляет динамический массив, то для него в процессе работы должна выделяться память. При добавлении в список новой строки память для нее выделяется автоматически. Свойство

```
property Capacity: Integer;
```

определяет число строк, для которых выделена память. Вы можете самостоятельно управлять этим параметром, помня при этом, что значение `Capacity` всегда должно быть больше или равно значению `Count`.

Свойство

```
property Duplicates: TDuplicates;
```

определяет, можно ли добавлять в список повторные значения.

Тип

```
type
  TDuplicates = (dupIgnore, dupAccept, dupError);
```

определяет реакцию списка на добавление повторного элемента:

- `dupIgnore` — запрещает добавление повторных элементов;
- `dupAccept` — разрешает добавление повторных элементов;
- `dupError` — запрещает добавление повторных элементов и генерирует исключительную ситуацию.

Класс `TStringList` трудно представить себе без возможностей сортировки. Если вас удовлетворит обычная сортировка, то для этого можно использовать свойство `Sorted` (сортировка выполняется при значении `True`) или метод `Sort`. Под "обычной" имеется в виду сортировка по тексту строк функцией `AnsiCompareStr` (т. е. с учетом национальных символов, в порядке возрастания). Если вы хотите отсортировать список по другому критерию, к вашим услугам метод:

```
type
  TStringListSortCompare = function(List: TStringList; Index1,
Index2: Integer): Integer;
procedure CustomSort(Compare: TStringListSortCompare);
```

Чтобы отсортировать список, вы должны описать функцию сравнения двух элементов с индексами `Index1` и `Index2`, которая должна возвращать следующие результаты:

- −1, если элемент с индексом `Index1` вы хотите поместить впереди элемента `Index2`;
- 0, если они равны;
- 1, если элемент с индексом `Index1` вы хотите поместить после элемента `Index2`.

Для описанного примера с объектом-городом нужны три процедуры (листинг 10.1).

Листинг 10.1. Пример сортировки списка

```
function SortByStatus(List: TStringList; Index1, Index2: Integer):
Integer;
begin
Result := AnsiCompareStr( (List.Objects[Index1] as TCityProps).Status,
  (List.Objects[Index2] as TCityProps).Status;
end;
function SortBySquare(List: TStringList; Index1, Index2: Integer):
Integer;
begin
 if (List.Objects[Index1] as TCityProps).Square <
  (List.Objects[Index2] as TCityProps). Square) then Result := -1
 else if (List.Objects[Index1] as TCityProps).Square =
  (List.Objects[Index2] as TCityProps).Square then Result := 0
 else Result := 1;
end;
function SortByPopulation(List: TStringList; Index1, Index2: Integer):
Integer;
begin
 if (List.Objects[Index1] as TCityProps).Population <
  (List.Objects[Index2] as TCityProps). Population then Result := -1
  else
  if (List.Objects[Index1] as TCityProps). Population =
   (List.Objects[Index2] as TCityProps). Population
   then Result := 0
   else Result := 1;
end;
```

Далее передаем одну из них в метод `CustomSort`:

```
Cities.CustomSort( SortByPopulation );
```

Для поиска нужного элемента используется метод

```
function Find(const S: string; var Index: Integer): Boolean;
```

В параметре `S` передается значение для поиска. В случае успеха функция возвращает `True`, а в параметре `Index` содержится индекс найденного элемента.

Метод

```
function IndexOf(const S: string): Integer;
```

возвращает индекс найденного элемента `S`. Иначе функция возвращает −1.

Метод

```
function IndexOfName(const Name: string): Integer;
```

возвращает индекс найденного элемента, для которого свойство `Names` совпадает со значением параметра `Name`.

Для поиска связанных объектов предусмотрен метод

```
function IndexOfObject(AObject: TObject): Integer;
```

В качестве параметра `AObject` должна передаваться ссылка на искомый объект.

А свойство

```
property CaseSensitive: Boolean;
```

включает или отключает режим поиска и сортировки с учетом регистра символов.

Помимо свойства `Strings`, содержимое списка можно получить при помощи свойств

```
property Text: string;
```

и

```
property CommaText: string;
```

Они представляют все строки списка в виде одной строки. При этом в первом свойстве элементы списка разделены символами возврата каретки и переноса строки. Во втором свойстве строки заключены в двойные кавычки и разделены запятыми или пробелами. Так, для списка городов (Москва, Петербург, Одесса) свойство `Text` будет равно

```
Москва#$D#$APетербург#$D#$AОдесса
```

а свойство `CommaText` равно

```
"Москва", "Петербург", "Одесса".
```

Важно иметь в виду, что эти свойства доступны не только по чтению, но и по записи. Так что заполнить список вы сможете не только циклически, вызывая методы `Add` или `Insert`, но и одним-единственным присвоением значения свойствам `Text` или `CommaText`.

Список может взаимодействовать с другими экземплярами класса `TStringList`.

Широко распространненый метод

```
procedure Assign(Source: TPersistent);
```

полностью переносит список `Source` в данный.

Метод

```
function Equals(Strings: TStrings): Boolean;
```

возвращает `True`, если элементы списка `Strings` полностью совпадают с элементами данного списка.

Список можно загрузить из файла или потока. Для этого существуют методы

```
procedure LoadFromFile(const FileName: string);
```

и

```
procedure LoadFromStream(Stream: TStream);
```

Сохранение списка выполняется методами

```
procedure SaveToFile(const FileName: string);
```

и

```
procedure SaveToStream(Stream: TStream);
```

Перед изменением списка вы можете получить управление, описав обработчик события

```
property OnChange: TNotifyEvent;
```

а после изменения

```
property OnChanging: TNotifyEvent;
```

На диске, прилагаемом к этой книге вы можете ознакомиться с примером работы со списком строк DemoStrings.

Список указателей

Для хранения списка указателей на размещенные в адресном пространстве структуры (объекты, динамические массивы, переменные) предназначен класс `TList`. Так же как и список строк `TStringList`, список указателей обеспечивает эффективную работу с элементами списка.

Класс *TList*

Основой класса `TList` является список указателей. Сам список представляет собой динамический массив указателей, к которому можно обратиться через индексированное свойство

```
property Items[Index: Integer]: Pointer;
```

Нумерация элементов начинается с нуля.

Прямой доступ к элементам массива возможен через свойство

```
type
  PPointerList = ^TPointerList;
  TPointerList = array[0..MaxListSize-1] of Pointer;
property List: PPointerList;
```

которое имеет атрибут "только для чтения".

Так как элементы списка являются указателями на некоторые структуры, прямое обращение к составным частям этих структур через свойство `Items` невозможно. Как это можно сделать, рассказывается далее в примере.

Примечание

В списке могут содержаться указатели на разнородные структуры. Не обязательно хранить в списке только указатели на объекты или указатели на записи.

Реализованные в классе `TList` операции со списком обеспечивают потребности разработчика и совпадают с операциями списка строк.

Добавить к концу списка новый указатель можно методом

```
function Add(Item: Pointer): Integer;
```

Прямое присваивание значения элементу, который еще не создан при помощи метода `Add`, вызовет ошибку времени выполнения.

Новый указатель можно добавить в нужное место списка с помощью метода

```
procedure Insert(Index: Integer; Item: Pointer);
```

В параметре `Index` указывается нужный порядковый номер в списке.

Перенос существующего элемента на новое место осуществляется методом

```
procedure Move(CurIndex, NewIndex: Integer);
```

Параметр `CurIndex` определяет старое положение указателя. Параметр `NewIndex` задает новое его положение.

Также можно поменять местами два элемента, определяемые параметрами `Index1` и `Index2`:

```
procedure Exchange(Index1, Index2: Integer);
```

Для удаления указателей из списка есть два метода. Если известен индекс, используется метод

```
procedure Delete(Index: Integer);
```

Если известен сам указатель, применяется метод

```
function Remove(Item: Pointer): Integer;
```

Эти методы не уменьшают объем памяти, выделенной под список. При необходимости сделать это надо задействовать свойство `Capacity`. Также существует метод `Expand`, который увеличивает отведенную память автоматически, в зависимости от текущего размера списка.

```
function Expand: TList;
```

Для того чтобы метод сработал, необходимо, чтобы `Count = Capacity`. Алгоритм работы метода представлен в табл. 10.1.

Таблица 10.1. *Алгоритм увеличения памяти списка*

Значение свойства `Capacity`	**На сколько увеличится свойство `Capacity`**
<4	4
4 .. 8	8
>8	16

Метод

```
procedure Clear; dynamic;
```

удаляет все элементы списка сразу.

Для поиска указателя по его значению служит метод

```
function IndexOf(Item: Pointer): Integer;
```

Метод возвращает индекс найденного элемента в списке. При неудачном поиске возвращается −1.

Для сортировки элементов списка применяется метод

```
type TListSortCompare = function (Item1, Item2: Pointer): Integer;
procedure Sort(Compare: TListSortCompare);
```

Так как состав структуры, на которую указывает элемент списка, невозможно заранее обобщить, разработка процедуры, осуществляющей сортировку, возлагается на программиста. Метод `Sort` лишь обеспечивает попарное сравнение указателей на основе созданного программистом алгоритма (пример сортировки был приведен в листинге 10.1).

Полностью все свойства и методы класса `TList` представлены в табл. 10.2.

Таблица 10.2. *Свойства и методы класса* `TList`

Объявление	Описание
`property Capacity: Integer;`	Определяет число строк, для которых выделена память
`property Count: Integer;`	Возвращает число строк в списке
`property Items[Index: Integer]: Pointer;`	Список указателей

`Type`
`TPointerList = array[0..MaxListSize-1] of Pointer;`
`PPointerList = ^TPointerList;`
`property List: PPointerList;`
	Динамический массив указателей

`function Add(Item: Pointer): Integer;`	Добавляет к списку новый указатель
`procedure Clear; dynamic;`	Полностью очищает список
`procedure Delete(Index: Integer);`	Удаляет указатель с индексом `Index`
`class procedure Error(const Msg: string; Data: Integer); virtual;`	Генерирует исключительную ситуацию `EListError`. Сообщение об ошибке создается из форматирующей строки `Msg` и числового параметра `Data`
`procedure Exchange(Index1, Index2: Integer);`	Меняет местами указатели с индексами `Index1` и `Index2`
`function Expand: TList;`	Увеличивает размер памяти, отведенной под список
`function First: Pointer;`	Возвращает первый указатель из списка
`function IndexOf(Item: Pointer): Integer;`	Возвращает индекс указателя, заданного параметром `Item`
`procedure Insert(Index: Integer; Item: Pointer);`	Вставляет новый элемент `Item` в позицию `Index`
`function Last: Pointer;`	Возвращает последний указатель в списке
`procedure Move(CurIndex, NewIndex: Integer);`	Перемещает элемент списка на новое место
`procedure Pack;`	Удаляет из списка все пустые (`Nil`) указатели

Таблица 10.2 (окончание)

Объявление	Описание
`function Remove(Item: Pointer): Integer;`	Удаляет из списка указатель `Item`
`type TListSortCompare = function (Item1, Item2: Pointer): Integer;` `procedure Sort(Compare: TListSortCompare);`	Сортирует элементы списка

Пример использования списка указателей

Рассмотрим работу со списками указателей на примере приложения DemoList. При щелчке мышью на форме приложения отображается точка, которой присваивается порядковый номер. Одновременно координаты и номер точки записываются в соответствующие свойства создаваемого экземпляра класса `TMyPixel`. Указатель на этот объект передается в новый элемент списка `PixList`.

В результате, после очистки формы всю последовательность точек можно восстановить, использовав указатели на объекты точек из списка.

Список точек можно отсортировать по координате X в порядке возрастания. Модуль главной формы проекта DemoList содержится в листинге 10.2.

Листинг 10.2. Модуль главной формы проекта DemoList

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Buttons;

type
  TMainForm = class(TForm)
    ListBtn: TBitBtn;
    ClearBtn: TBitBtn;
    DelBtn: TBitBtn;
    SortBtn: TBitBtn;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
```

```
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure ListBtnClick(Sender: TObject);
    procedure ClearBtnClick(Sender: TObject);
    procedure DelBtnClick(Sender: TObject);
    procedure SortBtnClick(Sender: TObject);
  private
    PixList: TList;
    PixNum: Integer;
  public
    { Public declarations }
  end;
  TMyPixel = class(TObject)
    FX:     Integer;
    FY:     Integer;
    FText:  Integer;
    constructor Create(X, Y, Num: Integer);
    procedure SetPixel;
  end;
var
  MainForm: TMainForm;
implementation
{$R *.DFM}
const PixColor = clRed;
var CurPixel: TMyPixel;
constructor TMyPixel.Create(X, Y, Num: Integer);
begin
 inherited Create;
 FX := X;
 FY := Y;
 FText := Num;
 SetPixel;
end;
procedure TMyPixel.SetPixel;
begin
 MainForm.Canvas.PolyLine([Point(FX, FY), Point(FX, FY)]);
 MainForm.Canvas.TextOut(FX + 1, FY + 1, IntToStr(FText));
end;
```

```
function PixCompare(Item1, Item2: Pointer): Integer;
var Pix1, Pix2: TMyPixel;
begin
 Pix1 := Item1;
 Pix2 := Item2;
 Result := Pix1.FX – Pix2.FX;
end;
procedure TMainForm.FormCreate(Sender: TObject);
begin
 PixList := TList.Create;
 PixNum := 1;                    {Счетчик точек}
 Canvas.Pen.Color := PixColor; {Цвет точки}
 Canvas.Pen.Width := 3;          {Размер точки}
 Canvas.Brush.Color := Color;  {Цвет фора текста равен цвету формы}
end;
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 PixList.Free;
end;
procedure TMainForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 PixList.Add(TMyPixel.Create(X, Y, PixNum));
 Inc(PixNum);
end;
procedure TMainForm.ListBtnClick(Sender: TObject);
var i: Integer;
begin
 with PixList do
 for i := 0 to Count – 1 do
 begin
  CurPixel := Items[i];
  CurPixel.SetPixel;
 end;
end;
procedure TMainForm.ClearBtnClick(Sender: TObject);
begin
 Canvas.FillRect(Rect(0, 0, Width, Height));
end;
```

```
procedure TMainForm.DelBtnClick(Sender: TObject);
begin
 PixList.Clear;
 PixNum := 1;
end;
procedure TMainForm.SortBtnClick(Sender: TObject);
var i: Integer;
begin
 PixList.Sort(PixCompare);
 with PixList do
 for i := 0 to Count — 1 do TMyPixel(Items[i]).FText := i + 1;
end;
end.
```

Класс `TMyPixel` обеспечивает хранение координат точки и ее порядковый номер в серии. Эти параметры передаются в конструктор класса. Метод `SetPixel` обеспечивает отрисовку точки на канве формы *(см. главу 12)*.

Экземпляр класса создается для каждой новой точки при щелчке кнопкой мыши в методе-обработчике `FormMouseDown`. Здесь же указатель на новый объект сохраняется в создаваемом при помощи метода `Add` элементе списка `PixList`. Таким образом, программа "запоминает" расположение и порядок следования точек.

Метод-обработчик `ListBtnClick` обеспечивает отображение точек. Для этого в цикле текущий указатель списка передается в переменную объектного типа `CurPixel`. Следовательно, в этой переменной то очереди "побывают" все созданные объекты, указатели на которые хранятся в списке.

Это сделано для того, чтобы получить доступ к свойствам объектов (непосредственно через указатель этого сделать нельзя). Второй способ приведения типа рассмотрен в методе-обработчике `SortBtnClick`.

Перед вторичным отображением точек необходимо очистить поверхность формы. Эту операцию выполняет метод-обработчик `ClearBtnClick`.

Список точек можно отсортировать по координате *X* в порядке возрастания. Для этого в методе-обработчике `SortBtnClick` вызывается метод `Sort` списка `PixList`. В параметре метода (переменная процедурного типа) передается функция `PixCompare`, которая обеспечивает инкапсулированный в методе `Sort` механизм перебора элементов списка алгоритмом принятия решения о старшинстве двух соседних элементов.

Если функция возвращает положительное число, то элемент `Item1` больше элемента `Item2`. Если результат отрицательный, то `Item1` меньше чем `Item2`. Если элементы равны, функция должна возвращать ноль.

В нашем случае сравнивались координаты *X* двух точек. В результате такой сортировки по возрастанию объекты оказались расположены так, что первый элемент списка указывает на объект с минимальной координатой *X*, а последний — с максимальной.

После сортировки осталось заново пронумеровать все точки. Это делает цикл в методе-обработчике `SortBtnClick`. Обратите внимание на примененный в этом случае способ приведения типа, обеспечивающий обращение к свойствам экземпляров класса `TMyPixel`.

Метод-обработчик `DelBtnClick` обеспечивает полную очистку списка `PixList`.

Коллекции

Коллекция представляет собой разновидность списка указателей, оптимизированную для работы с объектами определенного вида. Сама коллекция инкапсулирована в классе `TCollection`. Элемент коллекции должен быть экземпляром класса, унаследованного от класса `TCollectionItem`. Это облегчает программирование и позволяет обращаться к свойствам и методам объектов напрямую.

Коллекции объектов широко используются в компонентах VCL. Например, панели компонента `TCoolBar` *(см. главу 8)* объединены в коллекцию. Класс `TCoolBands`, объединяющий панели, является наследником класса `TCollection`. А отдельная панель является экземпляром класса `TCoolBar`, происходящего от класса `TCollectionItem`.

Поэтому знание свойств и методов классов коллекции позволит успешно использовать их при работе со многими компонентами (`TDBGrid`, `TListView`, `TStatusBar`, `TCoolBar` и т. д.).

Для работы с коллекцией, независимо от инкапсулирующего ее компонента, применяется специализированный Редактор коллекции (рис. 10.1), набор элементов управления которого может немного изменяться для разных компонентов.

Список Редактора объединяет элементы коллекции. При выборе одной строки из списка свойства объекта коллекции становятся доступны в Инспекторе объектов. В список можно добавлять новые элементы и удалять существующие, а также менять их взаимное положение.

Примеры использования коллекций представлены при описании соответствующих компонентов.

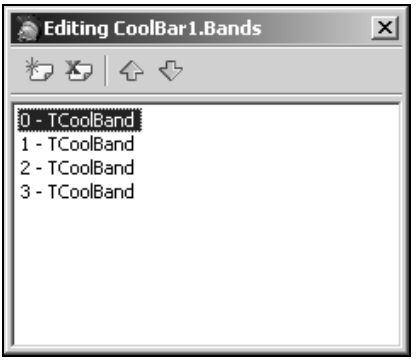


Рис. 10.1. Редактор коллекции

Класс *TCollection*

Класс `TCollection` является оболочкой коллекции, обеспечивая разработчика набором свойств и методов для управления ею.

Сама коллекция содержится в свойстве

```
property Items[Index: Integer]: TCollectionItem;
```

Полное объявление свойства в классе выглядит следующим образом:

```
property Items[Index: Integer]: TCollectionItem read GetItem write
SetItem;
```

Методы `GetItem` и `SetItem` обращаются к внутреннему полю `FItems`:

```
FItems: TList;
```

Именно оно хранит коллекцию объектов во время выполнения. Отсюда следует, что коллекция представляет собой список указателей на экземпляры класса `TCollectionItem` или его наследника. А класс `TCollection` обеспечивает удобство использования элементов списка. Свойства и методы класса `TCollection` приведены в табл. 10.3.

Таблица 10.3. *Свойства и методы класса* `TCollection`

Объявление	Описание
`Property Count: Integer;`	Возвращает число элементов коллекции
`Type TcollectionItemClass = class of TcollectionItem;` `Property ItemClass: TcollectionItemClass;`	Возвращает класс-наследник `TCollectionItem`, экземпляры которого собраны в коллекции

Таблица 10.3 *(окончание)*

Объявление	Описание
`Property Items[Index: Integer]: TcollectionItem;`	Коллекция экземпляров класса
`Function Add: TcollectionItem;`	Добавляет к коллекции новый экземпляр класса
`Procedure Assign(Source: TPersistent); override;`	Копирует коллекцию из объекта `Source` в данный объект
`Procedure BeginUpdate; virtual;`	Отменяет перерисовку коллекции. Используется при внесении изменений в коллекцию
`Procedure Clear;`	Удаляет из коллекции все элементы
`Procedure EndUpdate; virtual;`	Отменяет действие метода `BeginUpdate`
`Function FindItemID(ID: Integer): TCollectionItem;`	Возвращает объект коллекции с номером `ID`
`Function GetNamePath: string; override;`	Возвращает имя класса коллекции во время выполнения, если коллекция не имеет владельца. Иначе возвращает название свойства класса, владеющего коллекцией
`Function Insert(Index: Integer): TcollectionItem;`	Вставляет в коллекцию новый объект на место с номером `Index`

Класс *TCollectionItem*

Класс `TCollectionItem` инкапсулирует основные свойства и методы элемента коллекции. Свойства класса обеспечивают хранение информации о расположении элемента в коллекции (табл. 10.4).

Таблица 10.4. *Свойства и методы класса* `TCollectionItem`

Объявление	Описание
`Property Collection: Tcollection;`	Содержит экземпляр класса коллекции, которой принадлежит данный элемент
`Property DisplayName: string;`	Содержит имя элемента, которое представляет его в Редакторе коллекции
`property ID: Integer;`	Содержит уникальный номер элемента в коллекции, который не может изменяться
`property Index: Integer;`	Содержит порядковый номер элемента в коллекции. Он соответствует положению элемента в списке и может изменяться

Резюме

Списки, объединяющие элементы различных типов, играют важную роль при создании программной логики приложения. В Delphi есть три основных вида списков.

Классы `TStrings` и `TStringList` обеспечивают применение списков строк.

Класс `TList` инкапсулирует список указателей.

Классы `TCollection` и `TCollectionItem` позволяют применять в компонентах и программном коде коллекции — группы однородных объектов.

В среде Delphi вы можете найти еще много полезных классов общего применения. В модуле CLASSES.PAS есть класс `TBits`, обеспечивающий побитовое чтение и запись информации. В модуле CONTNRS.PAS есть классы `TStack` и `TQueue` (стек и очередь), а также потомки `TList` — `TClassList`, `TComponentList` и т. д. Они помогут вам решать типовые задачи быстро и не "изобретая велосипед".

Глава 11

Файлы и потоки

Большинство приложений создаются для того, чтобы обрабатывать данные — это прописная истина. С развитием программных технологий мы получаем возможность получать и посылать все более крупные и сложные массивы данных; однако до сих пор 90% из них хранятся в файлах.

Для работы с файлами в приложении разработчику приходится решать множество задач. Основные из них — это поиск необходимого файла и выполнение с ним операций ввода-вывода.

Основные принципы и структура файловой системы мало изменились со времен MS-DOS. Файловые системы (FAT32, NTFS), появившаяся в Windows 2000 служба Active Directory, не изменяют главного — понятия файла и способов обращения к нему.

Среда Delphi дает вам возможность выбрать один из четырех вариантов работы.

1. Традиционный набор функций работы с файлами, унаследованный от Turbo Pascal.
2. Функции ввода/вывода из Windows API.
3. Потоки (`TStream` и его потомки).
4. Отображаемые файлы.

В этой главе мы изучим все основные способы работы с файлами в приложениях Delphi на конкретных примерах создания программного кода.

Использование файловых переменных. Типы файлов

Современный программный код Delphi для чтения данных из файла зачастую удивительно похож на аналогичный, написанный к примеру в Turbo Pascal 4.0. Это возможно потому, что программисты Borland сохранили неизменным "старый добрый" набор файловых функций, работающих через файловые переменные.

При организации операций файлового ввода-вывода в приложении большое значение имеет род информации, содержащейся в файле. Чаще всего это строки, но встречаются двоичные данные или структурированная информация, например, массивы или записи.

Естественно, что сведения о типе хранящихся в файле данных важно изначально задать. Для этого предусмотрены специальные файловые переменные, определяющие тип файла. Они делятся на нетипизированные и типизированные.

Перед началом работы с любым файлом необходимо описать файловую переменную, соответствующую типу данных этого файла. В дальнейшем эта переменная используется при обращении к файлу.

В Delphi имеется возможность создавать нетипизированные файлы. Для их обозначения употребляется ключевое слово `file`:

```
var UntypedFile: file;
```

Такие файловые переменные служат для организации быстрого и эффективного ввода-вывода безотносительно к типу данных. При этом подразумевается, что данные читаются или записываются в виде двоичного массива. Для этого имеются специальные процедуры блочного чтения и записи.

Типизированные файлы обеспечивают ввод-вывод с учетом конкретного типа данных. Их объявляют ключевым словом `file of`, к которому добавляется конкретный тип данных. Например, для работы с файлом, содержащим набор байтов, файловая переменная объявляется так:

```
var ByteFile: file of byte;
```

При этом допустимы любые типы фиксированного размера, за исключением указателей. Разрешается использовать структурные типы, если их составные части удовлетворяют названному выше ограничению. Например, можно создать файловую переменную для записи:

```
type Country = record
      Name: String;
      Capital: String;
      Population: LongInt;
      Area: LongInt;
     end;
var CountryFile: file of Country;
```

Для работы с текстовыми файлами есть специальная файловая переменная `TextFile` или `Text`:

```
var F: TextFile;
```

Операции ввода-вывода

Теперь рассмотрим две самые распространенные операции, выполняемые при работе с файлами. Это чтение и запись. Для их выполнения применяются специальные функции файлового ввода-вывода.

Итак, для выполнения операции чтения или записи необходимо выполнить следующие действия.

1. Объявить файловую переменную необходимого типа.
2. При помощи функции `AssignFile` связать эту переменную с требуемым файлом.
3. Открыть файл при помощи функций `Append`, `Reset`, `Rewrite`.
4. Выполнить операции чтения или записи. При этом в зависимости от сложности задачи и структуры данных может использоваться целый ряд вспомогательных функций.
5. Закрыть файл при помощи функции `CloseFile`.

В качестве примера рассмотрим небольшой фрагмент исходного кода (листинг 11.1).

Листинг 11.1. Пример работы с файлом

```
...
var F: TextFile;
    S: string;
begin
 if OpenDlg.Execute
  then AssignFile(F, OpenDlg.FileName)
  else Exit;
 Reset(F);
 while Not EOF(F) do
 begin
  Readln(F, S);
  Memo.Lines.Add(S);
 end;
 CloseFile(F);
end;
...
```

Если в диалоге открытия файла `OpenDlg` был выбран файл, то его имя связывается с файловой переменной `F` при помощи процедуры `AssignFile`. Ре-

комендуется указывать всегда полное имя файла (включая его маршрут). Как раз в таком виде возвращают результат выбора файла диалоги работы с файлами `TOpenDialog` и `TOpenPictureDialog`. Затем при помощи процедуры `Reset` этот файл открывается для чтения и записи.

В цикле осуществляются чтение из файла текстовых строк и запись их в компонент `TMemo`. Процедура `Readln` осуществляет чтение текущей строки файла и переходит на следующую строку. Цикл выполняется, пока функция `EOF` не сообщит о достижении конца файла.

После завершения чтения файл закрывается.

Такой же исходный код пригоден и для записи данных в файл. Необходимо только заменить процедуру чтения на процедуру записи.

Теперь остановимся подробнее на назначении функций для файлового ввода-вывода.

Открытие файла может осуществляться тремя процедурами — в зависимости от типа его дальнейшего использования.

Процедура

```
procedure Reset(var F [: File; RecSize: Word ] );
```

открывает существующий файл для чтения и записи, текущая позиция устанавливается на первой строке файла.

Процедура

```
procedure Append(var F: Text);
```

открывает файл для записи информации после его последней строки, текущая позиция устанавливается на конец файла.

Процедура

```
procedure Rewrite(var F: File [; Recsize: Word ] );
```

создает новый файл и открывает его, текущая позиция устанавливается в начало файла. Если файл с таким именем уже существует, то он перезаписывается.

Переменная `RecSize` необходима только при работе с нетипизированными файлами и определяет размер одной записи для операции передачи данных. Если данный параметр опущен, то по умолчанию `RecSize` равно 128 байт.

Чтение данных из типизированных и текстовых файлов выполняют процедуры `Read` и `Readln`.

Процедура `Read` имеет различное объявление для текстовых и других типизированных файлов:

- `procedure Read([var F: Text;] V1 [, V2,...,Vn]);` — для текстовых файлов;

- `procedure Read(F, V1 [, V2,...,Vn]);` — для других типизированных файлов.

При одном вызове процедуры можно читать данные в произвольное число переменных. Естественно, что тип переменных должен совпадать с типом файла. При чтении в очередную переменную читается ровно столько байтов из файла, сколько занимает тип данных. В следующую переменную читается столько же байтов, расположенных следом. После выполнения процедуры текущая позиция устанавливается на первом непрочитанном байте. Аналогично работают несколько процедур `Read` для одной переменной, выполненных подряд.

Процедура

```
procedure Readln([ var F: Text; ] V1 [, V2, ...,Vn ]);
```

считывает одну сроку текстового файла и устанавливает текущую позицию на следующей строке. Если вызвать процедуру без переменных `V1..Vn`, то она просто передвигает текущую позицию на следующую строку файла.

Процедуры для записи в файлы `Write` и `Writeln` описаны аналогично:

```
procedure Write( [var F: Text; ] P1 [ , P2,..., Pn] );
procedure Writeln([ var F: Text; ] P1 [, P2, ...,Pn ] );
```

Параметры `P1, P2, ... , Pn` могут быть одним из целых или вещественных типов, одним из строковых типов или логическим типом. Но у них есть возможность дополнительного форматирования при выводе. Каждый параметр записи может иметь имеет форму:

```
Pn [: MinWidth [: DecPlaces ] ]
```

Здесь

- `Pn` — выводимая переменная или выражение;
- `MinWidth` — минимальная ширина поля в символах, которая должна быть больше 0;
- `DecPlaces` — содержит количество десятичных символов после запятой при отображении вещественных чисел с фиксированной точкой.

Обратите внимание, что для текстовых файлов в функциях `Read` и `Write` файловая переменная `F` может быть опущена. В этом случае чтение и запись осуществляются в стандартные файлы ввода-вывода. Когда программа компилируется как консольное приложение (флаг `{$APPTYPE CONSOLE}`), Delphi автоматически связывает входной и выходной файлы с окном консоли.

Для контроля за текущей позицией в файле применяются две основные функции. Функция `EOF(F)` возвращает значение `True`, если достигнут конец файла, а `EOLN(F)` аналогично сигнализирует о достижении конца строки.

Естественно, в качестве параметра в функции необходимо передавать файловую переменную.

Процедура

```
procedure Seek(var F; N: Longint);
```

обеспечивает смещение текущей позиции на `N` элементов. Размер одного элемента в байтах зависит от типа данных файла (от типизированной переменной).

Рассмотрим теперь режим блочного ввода-вывода данных между файлом и областью адресного пространства (буфером). Этот режим отличается значительной скоростью передачи данных, причем скорость пропорциональна размеру одного передаваемого блока — чем больше блок, тем больше скорость.

Для реализации данного режима допустимы только нетипизированные файловые переменные. Размер блока определяется в процедуре открытия файла (`Reset`, `Rewrite`). Непосредственно для выполнения операций применяются процедуры `BlockRead` и `BlockWrite`.

Процедура

```
procedure BlockRead(var F: File; var Buf; Count: Integer [;
var AmtTransferred: Integer]);
```

выполняет запись блока из файла в буфер. Параметр `F` ссылается на нетипизированную файловую переменную, связанную с нужным файлом.

Параметр `Buf` определяет любую переменную (число, строка, массив, структура), в которую читаются байты из файла. Параметр `Count` определяет число считываемых блоков. Наконец, необязательный параметр `AmtTransferred` возвращает число реально считанных блоков.

При блочном чтении или записи размер блока необходимо выбирать таким образом, чтобы он был кратен размеру одного значения того типа, который хранится в файле. Например, если в файле хранятся значения типа `Double` (8 байт), то размер блока может быть равен 8, 16, 24, 32 и т. д. Фрагмент исходного кода блочного чтения из такого файла представлен в листинге 11.2.

Листинг 11.2. Пример блочного чтения из файла

```
...
var F:           File;
    DoubleArray: array [0..255] of Double;
    Transfered:  Integer;
begin
 if OpenDlg.Execute then AssignFile(F, OpenDlg.FileName) else Exit;
 Reset(F, 64);
```

```
 BlockRead(F, DoubleArray, 32, Transfered);
 CloseFile(F);
 ShowMessage('Считано '+IntToStr(Transfered)+' блоков');
end;
...
```

Как видно из примера, размер блока установлен в процедуре `Reset` и кратен размеру элемента массива `DoubleArray`, в который считываются данные. В переменной `Transfered` возвращается число считанных блоков. Если размер файла меньше заданного в процедуре `BlockRead` числа блоков, ошибка не возникает, а в переменной `Transfered` передается число реально считанных блоков.

Процедура

```
procedure BlockWrite(var f: File; var Buf; Count: Integer [; var
AmtTransferred: Integer]);
```

функционирует аналогично.

Оставшиеся функции ввода-вывода, работающие с файловыми переменными, подробно описаны в табл. 11.1.

Таблица 11.1. *Процедуры и функции для работы с файлами*

Объявление	Описание
`function ChangeFileExt(const FileName, Extension: string): string;`	Функция позволяет изменить расширение файла. При этом сам файл не переименовывается
`Procedure ChDir(S: string);`	Процедура изменяет текущий каталог на другой, путь к которому описан в строке `S`
`procedure CloseFile(var F);`	Вызов процедуры разрывает связь между файловой переменной и файлом на диске
`function DeleteFile(const FileName: string): Boolean;`	Функция производит удаление файла `FileName` с диска и возвращает значение `False`, если файл удалить не удалось или файл не существует
`function ExtractFileExt(const FileName: string): string;`	Функция возвращает расширение файла
`function ExtractFileName(const FileName: string): string;`	Извлекает имя и расширение файла, содержащегося в параметре `FileName`
`function ExtractFilePath(const FileName: string): string;`	Функция возвращает полный путь к файлу

Таблица 11.1 *(продолжение)*

Объявление	Описание
`procedure Erase(var F);`	Удаляет файл, связанный с файловой переменной `F`
`function FileSearch(const Name, DirList: string): string;`	Данная процедура осуществляет поиск в каталогах `DirList` файла `Name`. Если в процессе выполнения `FileSearch` обнаруживается искомое имя файла, то функция возвращает в строке типа `String` полный путь к найденному файлу. Если файл не найден, то возвращается пустая строка
`function FileSetAttr(const FileName: string; Attr: Integer): Integer;`	Присваивает файлу с именем `FileName` атрибуты `Attr`. Функция возвращает 0, если присвоение атрибутов прошло успешно. В противном случае возвращается код ошибки
`function FilePos(var F): Longint;`	Возвращает текущую позицию файла. Функция используется для нетекстовых файлов. Перед вызовом `FilePos` файл должен быть открыт
`function FileSize(var F): Integer;`	`FileSize` возвращает размер файла в байтах или количество записей в файле, содержащем записи. Перед вызовом данной функции файл должен быть открыт. Для текстовых файлов функция `FileSize` не используется
`procedure Flush(var F: Text);`	Процедура очищает буфер текстового файла открытого для записи. `F` — файловая переменная. Когда текстовый файл открыт для записи с использованием функций `Rewrite` или `Append`, `Flush` очищает выходной буфер, связанный с файлом. После выполнения данной процедуры все символы, которые направлены для записи в файл, будут гарантированно записаны в нем
`procedure GetDir(D: Byte; var S: string);`	Возвращает число, соответствующее диску, на котором содержится текущий каталог `S`. `D` может принимать одно из следующих значений: 0 — по умолчанию (текущий), 1 — A, 2 — B, 3 — C и так далее.

Таблица 11.1 *(продолжение)*

Объявление	Описание
	Процедура не генерирует код ошибки. Если имя диска в `D` оказывается ошибочным, то в строке `S` возвращается значение `X:\`, как в случае, если бы текущий каталог был на этом ошибочно указанном диске
`function IOResult: Integer;`	Функция возвращает статус последней произведенной операции ввода-вывода, если контроль ошибок выключен `{$I-}`
`procedure MkDir(S: string);`	Процедура создает новый каталог, который описывается в строке `S`
`procedure Rename(var F; NewName: string);`	Процедура изменяет имя файла, связанного с файловой переменной `F`. Переменная `NewName` является строкой типа `string` или типа `PChar` (если включена поддержка расширенного синтаксиса)
`procedure RmDir(S: string);`	Процедура удаляет пустой каталог, путь к которому задается в строке `S`. Если указанный каталог не существует или он не пустой, то возникает сообщение об ошибке ввода-вывода
`procedure Seek(var F; N: Longint);`	Перемещает текущую позицию курсора на `N` позиций. Данная процедура применяется только для открытых типизированных или нетипизироанных файлов для проведения чтения/записи с нужной позиции файла. Началу файла соответствует нулевой номер позиции. Для добавления новой информации в конец существующего файла необходимо установить указатель на символ, следующий за последним. Для этого можно использовать выражение `Seek(F, FileSize(F))`
`function SeekEof[(var F: Text)]: Boolean;`	Возвращает значение `True`, если указатель текущей позиции находится на символе конца файла. `SeekEof` может быть использован только с открытым текстовым файлом
`function SeekEoln[(var F: Text)]: Boolean;`	Возвращает значение `True`, если указатель текущей позиции находится на символе конца строки. `SeekEoln` может быть использован только с открытым текстовым файлом

Таблица 11.1 *(окончание)*

Объявление	Описание
`procedure SetTextBuf(var F: Text; var Buf [ ; Size: Integer]);`	Связывает с текстовым файлом буфер ввода-вывода. `F` — файловая переменная текстового типа. Каждая файловая переменная текстового типа имеет внутренний буфер емкостью 128 байт, в котором накапливаются данные при чтении и записи. Такой буфер пригоден для большинства операций. Однако при выполнении программ с интенсивным вводом-выводом, буфер может переполниться, что приведет к записи операций ввода-вывода на диск и, как следствие, к существенному замедлению работы приложения. `SetTextBuf` позволяет помещать в текстовый файл `F` информацию об операциях ввода-вывода вместо ее размещения в буфере. `Size` указывает размер буфера в байтах. Если этот параметр опускается, то полагается размер, равный `SizeOf(Buf)`. Новый буфер действует до тех пор, пока `F` не будет связана с новым файлом процедурой `AssignFile`
`procedure Truncate(var F);`	Удаляет все позиции, следующие после текущей позиции в файле. А текущая позиция становится концом файла. С переменной `F` может быть связан файл любого типа, за исключением текстового

Ввод-вывод с использованием функций Windows API

Для тех, кто знаком с языком C++ или Java, более привычными будут стандартные функции работы с файлами Windows. Тем более, что возможности ввода-вывода в них расширены. Каждый файл в этом наборе функций описывается не переменной, а дескриптором (`Handle`) — 32-разрядной величиной, которая идентифицирует файл в операционной системе.

В Win32 файл открывается при помощи функции, имеющей обманчивое название

```
function CreateFile(lpFileName: PChar; dwDesiredAccess, dwShareMode:
DWORD;   lpSecurityAttributes: PSecurityAttributes; dwCreationDistribution,
dwFlagsAndAttributes: DWORD; hTemplateFile: THandle): THandle;
```

Хоть ее название и начинается с `Create`, но она позволяет не только создавать, но и открывать уже существующие файлы.

Такое огромное количество параметров оправдано, т. к. `CreateFile` позволяет открывать файлы на диске, устройства, каналы, порты и вообще любые источники ввода-вывода. Назначение параметров описано в табл. 11.2.

Таблица 11.2. *Параметры функции* `CreateFile`

Параметр	Смысл
`LpFileName: pChar`	Имя открываемого объекта. Может представлять собой традиционную строку с путем и именем файла, UNC (для открытия объектов в сети, имя порта, драйвера или устройства
`DwDesiredAccess,: DWORD`	Способ доступа к объекту. Может быть равен: • `GENERIC_READ` — для чтения; • `GENERIC_WRITE` — для записи. Их комбинация позволяет открыть файл для чтения и записи. Параметр 0 применяется, если нужно получить атрибуты файла без его фактического открытия
`dwShareMode: DWORD`	Режим совместного использования файла: • 0 — совместный доступ запрещен; • `FILE_SHARE_READ` — для чтения; • `FILE_SHARE_WRITE` — для записи. Их комбинация — для полного совместного доступа
`lpSecurityAttributes: PSecurityAttributes`	Атрибуты защиты файла. В Windows 95/98 не используются (должны быть равны `nil`). В Windows NT/2000 этот параметр, равный `nil`, дает объекту атрибуты по умолчанию
`DwCreationDistribution: DWORD;`	Способ открытия файла: • `CREATE_NEW` — создается новый файл; если таковой уже существует, функция возвращает ошибку `ERROR_ALREADY_EXISTS`; • `CREATE_ALWAYS` — создается новый файл; если таковой уже существует, он перезаписывается;

***Таблица 11.2** (окончание)*

Параметр	Смысл
	• `OPEN_EXISTING` — открывает существующий файл, если таковой не найден, функция возвращает ошибку; • `OPEN_ALWAYS` — открывает существующий файл, если таковой не найден, он создается
`DwFlagsAndAttributes: DWORD;`	Набор атрибутов (скрытый, системный, сжатый) и флагов для открытия объекта. Подробное описание см. в документации по Win32
`hTemplateFile: THandle`	Файл-шаблон, атрибуты которого используются для открытия. В Windows 95/98 не применяется и должен быть равен 0

Эта функция возвращает дескриптор открытого объекта ввода-вывода. Если открытие невозможно из-за ошибок, возвращается код `INVALID_HANDLE_VALUE`, а расширенный код ошибки можно узнать, вызвав функцию `GetLastError`.

Закрывается файл в Win32 функцией `CloseHandle` (не `CloseFile`, а `CloseHandle`! Правда, "легко" запомнить? Что поделать, так их назвали разработчики Win32).

Приведем некоторые приемы работы с `CreateFile`. Часто программисты хотят иметь возможность организовать посекторный доступ к физическим устройствам хранения — например, к дискете. Листинг 11.3 иллюстрирует это.

Листинг 11.3. Чтение сектора с дискеты при помощи функции `CreateFile`

```
type
pDIOCRegs = ^TDIOCRegs;
TDIOCRegs = packed record
rEBX,rEDX,rECX,rEAX,rEDI, rESI, rFlags : DWORD;
end;
const VWIN32_DIOC_DOS_IOCTL = 1;
  VWIN32_DIOC_DOS_INT13 = 4;  // Performs Interrupt 13h commands.
  SectorSize = 512;
function ReadSector(Head, Track, Sector: Integer; buffer : pointer;
Floppy: char):Boolean;
var hDevice : THandle;
  Regs : TDIOCRegs;
```

```
  DevName : string;
  nb : Integer;
begin
 if WIN32PLATFORM <> VER_PLATFORM_WIN32_NT then
 begin {win95/98}
 hDevice := CreateFile('\\.\vwin32',  GENERIC_READ, 0, nil, 0,
 FILE_FLAG_DELETE_ON_CLOSE, 0);
 if (hDevice = INVALID_HANDLE_VALUE) then
begin
 Result := FALSE;
 Exit;
end;
 regs.rEDX := Head * $100 + Ord(Floppy in ['b', 'B']);
 regs.rEAX := $201; // код операции read sector
 regs.rEBX := DWORD(buffer); // buffer
 regs.rECX := Track * $100 + Sector;
 regs.rFlags := $0;
 Result := DeviceIoControl(hDevice,VWIN32_DIOC_DOS_INT13,
   @regs, sizeof(regs),      @regs, sizeof(regs), nb, nil)
  and ((regs.rFlags and  $1)=0);
 CloseHandle(hDevice);
 end {win95/98}
else
begin // Windows NT/2000
 DevName :='\\.\A:';
 if Floppy in ['b', 'B'] then DevName[5] := Floppy;
 hDevice := CreateFile(pChar(Devname), GENERIC_READ,
       FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL, 0);
  if (hDevice = INVALID_HANDLE_VALUE) then
 begin
  Result := FALSE;
 Exit;
end;
 SetFilePointer(hDevice, (Sector-1)*SectorSize, nil,
 FILE_BEGIN);{нумерация с 1 }
 Result := ReadFile(hDevice, buffer^, SectorSize, nb, nil) and
(nb=SectorSize);
 CloseHandle(hDevice);
 end; // Windows NT/2000
end;
```

Для чтения и записи данных в Win32 существуют функции

```
function ReadFile(hFile: THandle; var Buffer; nNumberOfBytesToRead:
DWORD; var lpNumberOfBytesRead: DWORD; lpOverlapped: POverlapped): BOOL;
function WriteFile(hFile: THandle; const Buffer; nNumberOfBytesToWrite:
DWORD; var lpNumberOfBytesWritten: DWORD; lpOverlapped: POverlapped):
BOOL;
```

Здесь все сходно с `BlockRead` и `BlockWrite`: `hFile` — это дескриптор файла, `Buffer` — адрес, по которому будут читаться (писаться) данные; третий параметр означает требуемое число читаемых (записываемых) байт, а четвертый — фактически прочитанное (записанное). Последний параметр — `lpOverlapped` — обсудим чуть позже.

Функция `CreateFile` служит и для доступа к портам ввода-вывода. Часто программисты сталкиваются с задачей: как организовать обмен данными с различными нестандартными устройствами, подключенными к параллельному или последовательному порту? В Win32 прямой доступ к портам запрещен, и приходится открывать их как файлы:

```
hCom := CreateFile('COM2', GENERIC_READ or GENERIC_WRITE,
                   0, NIL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, 0);
if hCom = INVALID_HANDLE_VALUE then
  begin
  raise EAbort.CreateFmt('Ошибка открытия порта: %d',[GetLastError]);
  end;
```

Здесь самое большое отличие от предыдущего примера — в скромном флаге `FILE_FLAG_OVERLAPPED`. О роли этих изменений — в следующем разделе.

Отложенный (асинхронный) ввод-вывод

Эта принципиально новая возможность введена впервые в Win32 с появлением реальной многозадачности. Вызывая функции чтения и записи данных, вы на самом деле передаете исходные данные одному из потоков (threads) операционной системы, который и осуществляет фактические обязанности по работе с устройством. Время доступа всех периферийных устройств гораздо больше доступа к ОЗУ, и ваша программа, вызвавшая `Read` или `Write`, будет дожидаться окончания операции ввода-вывода. Замедление работы программы налицо.

Выход был найден в реализации отложенного (overlapped) ввода-вывода. До начала отложенного ввода-вывода инициализируются дескриптор объекта типа события (функция `CreateEvent`) и структура типа `TOverlapped`. Вы вызываете функцию `ReadFile` или `WriteFile`, в которой последним парамет-

ром указываете на TOverlapped. Эта структура содержит дескриптор события Windows (event).

ОС начинает операцию (ее выполняет отдельный программный поток, скрытый от программиста) и немедленно возвращает управление; вы можете не тратить время на ожидание. Признак того, что операция началась и продолжается — возврат кода возврата ERROR_IO_PENDING. Пусть вас не пугает слово "error" в названии — это совершенно нормально. Если операция продолжается долго (а чтение и запись файлов на дискете, да и на диске, чтение/запись именованных каналов можно отнести к "длинным" операциям), то программа может спокойно выполнять последующие операторы. Событие будет "взведено" ОС тогда, когда ввод-вывод закончится.

Когда, по мнению программиста, ввод-вывод должен быть завершен, можно проверить это с помощью функции WaitForSingleObject.

```
function WaitForSingleObject(hHandle: THandle; dwMilliseconds: DWORD):
DWORD;
```

Объект ожидания (параметр hHandle) в этом случае — тот самый, который создан нами, указан в структуре TOverlapped и передан в качестве параметра в функцию ReadFile или WriteFile. Можно указать любое время ожидания, в том числе бесконечное (параметр TimeOut при этом равен константе INFINITE). Признаком нормального завершения служит получение кода возврата WAIT_OBJECT_0. Листинг 11.4 иллюстрирует пример отложенной операции чтения.

Листинг 11.4. Пример отложенной операции чтения

```
function TMyClass.Read(var Buffer; Count: Longint): Longint;
var succ : boolean;nb : Cardinal;LastError : Longint;
Overlap: TOverlapped;
begin
FillChar(Overlap,SizeOf(Overlap),0);
Overlap.hEvent := CreateEvent(nil, True, False, nil);
Result := MaxInt;
succ := ReadFile(FHandle, Buffer, Count, nb, @OverlapRd);
//
// здесь можно вставить любые операторы, которые
// могут быть выполнены до окончания ввода-вывода
//
 if not succ then
    begin
     LastError := GetLastError;
```

```
    if LastError = ERROR_IO_PENDING
     then
      begin
       if WaitForSingleObject(OverlapRd.hEvent, INFINITE)=WAIT_OBJECT_0
       then
         GetOverlappedResult(FHandle, OverlapRd, nb, TRUE);
      end
      else
       raise EAbort.Create(Format('Read failed, error %d',[LastError]));
    end;
 Result := nb;
CloseHandle( hEvent );
end;
```

Если вы задали конечный интервал в миллисекундах, а операция еще не закончена, `WaitForSingleObject` вернет код завершения `WAIT_TIMEOUT`. Функция `GetOverlappedResult` возвращает в параметре `nb` число байтов, действительно прочитанных или записанных во время отложенной операции.

Контроль ошибок ввода-вывода

При работе с файлами разработчик обязательно должен предусмотреть обработку возможных ошибок. Практика показывает, что именно операции ввода-вывода вызывают большую часть ошибок, возникающих в приложении из-за воздействия окружающей программной среды.

Контроль за ошибками ввода-вывода зависит от применяемых функций. При использовании доступа через Win32 API все функции возвращают код ошибки Windows, который и нужно проанализировать.

При возникновении ошибок ввода-вывода в функциях, работающих с файловыми переменными, генерируется исключительная ситуация класса `EInOutError`. Но так происходит только в том случае, если включен контроль ошибок ввода-вывода. Для этого предусмотрены соответствующие директивы компилятора:

- `{$I+}` — контроль включен (установлен по умолчанию);
- `{$I-}` — контроль отключен.

Класс `EInOutError` отличается тем, что у него есть поле `ErrorCode`. При возникновении этой исключительной ситуации вы можете получить его значение и принять решение.

Основные коды имеют такие значения:

- 2 — файл не найден;
- 3 — неверное имя файла;
- 4 — слишком много открытых файлов;
- 5 — доступ запрещен;
- 100 — достигнут конец файла;
- 101 — диск переполнен;
- 106 — ошибка ввода.

При отключенном контроле при возникновении ошибки выполнение программы продолжается без остановки. Однако в этом случае устранение возможных последствий ошибки возлагается на разработчика. Для этого применяется функция

```
function IOResult: Integer;
```

которая возвращает значение 0 при отсутствии ошибок.

Атрибуты файла. Поиск файла

Еще одна часто выполняемая с файлом операция — поиск файлов в заданном каталоге. Для организации поиска и отбора файлов применяются специальные процедуры, а также структура, в которой сохраняются результаты поиска.

Запись

```
type
TFileName = string;
TSearchRec = record
             Time: Integer;            {Время и дата создания}
             Size: Integer;            {Размер файла}
             Attr: Integer;            {Параметры файла}
             Name: TFileName;          {Полное имя файла}
             ExcludeAttr: Integer;     {Не используется}
             FindHandle: THandle;      {Дескриптор файла}
             FindData: TWin32FindData; {Не используется}
end;
```

обеспечивает хранение характеристик файла после удачного поиска. Дата и время создания файла хранятся в формате MS-DOS, поэтому для получения этих параметров в принятом в Delphi формате `TDateTime` необходимо использовать следующую функцию:

```
function FileDateToDateTime(FileDate: Integer): TDateTime;
```

Обратное преобразование выполняет функция

```
function DateTimeToFileDate(DateTime: TDateTime): Integer;
```

Свойство `Attr` может содержать комбинацию следующих флагов-значений:

- `faReadOnly` — только для чтения;
- `faHidden` — скрытый;
- `faSysFile` — системный;
- `faVolumeID` — метка тома;
- `faDirectory` — каталог;
- `faArchive` — архивный;
- `faAnyFile` — любой.

Для определения параметров файла служит оператор `AND`:

```
if (SearchRec.Attr AND faReadOnly) > 0
  then ShowMessage('Файл только для чтения');
```

Непосредственно для поиска файлов имеются функции `FindFirst` и `FindNext`.

Функция

```
function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec):
Integer;
```

находит первый файл, заданный полным маршрутом `Path` и параметрами `Attr`. Если заданный файл найден, функция возвращает 0, иначе — код ошибки Windows. Параметры найденного файла возвращаются в записи `F` типа `TSearchRec`.

Функция

```
function FindNext(var F: TSearchRec): Integer;
```

служит для повторного поиска следующего файла, удовлетворяющего критерию поиска. При этом учитываются те параметры поиска, которые заданы последним вызовом функции `FindFirst`. В случае удачного поиска возвращается 0.

Для освобождения ресурсов, выделенных для выполнения поиска, применяется функция:

```
procedure FindClose(var F: TSearchRec);
```

В качестве примера организации поиска файлов рассмотрим фрагмент исходного кода, в котором маршрут поиска файлов задается в однострочном текстовом редакторе `DirEdit`, а список найденных файлов передается в компонент `TListBox`.

```
procedure TForm1.FindBtnClick(Sender: TObject);
begin
```

```
 ListBox.Items.Clear;
 FindFirst(DirEdit.Text, faArchive + faHidden, SearchRec);
 while FindNext(SearchRec) = 0 do
 ListBox.Items.Add(SearchRec.Name);
 FindClose(SearchRec);
end;
```

Потоки

Потоки — очень удачное средство для унификации ввода-вывода для различных носителей. Потоки представляют собой специальные объекты — наследники абстрактного класса `TStream`. Сам `TStream` "умеет" открываться, читать, писать, изменять текущее положение и закрываться. Поскольку для разных носителей эти операции происходят по-разному, конкретные аспекты реализованы в его потомках. Наиболее часто потоки служат для работы с файлами на диске и памятью.

Многие классы VCL имеют унифицированные методы `LoadFromStream` и `SaveToStream`, которые обеспечивают обмен данными с потоками. От того, с каким физическим носителем работает поток, зависит место хранения данных.

Базовые классы *TStream* и *THandleStream*

В основе иерархии классов потоков лежит класс `TStream`. Он обеспечивает выполнение основных операций потока безотносительно к реальному носителю информации. Основными из них является чтение и запись данных.

Класс `TStream` порожден непосредственно от класса `TObject`.

Потоки также играют важную роль для чтения/записи компонентов из файлов ресурсов. Большая группа методов обеспечивает взаимодействие компонента и потока и чтение свойств компонента из ресурса и запись значений свойств в ресурс (табл. 11.3).

Таблица 11.3. *Свойства и методы класса* `TStream`

Объявление	Описание
`property Position: Longint;`	Определяет текущую позицию в потоке
`property Size: Longint;`	Определяет размер потока в байтах
`function CopyFrom(Source: TStream; Count: Longint): Longint;`	Копирует из потока `Source` `Count` байтов, начиная с текущей позиции. Возвращает число скопированных байтов

Таблица 11.3 *(окончание)*

Объявление	Описание
`function Read(var Buffer; Count: Longint): Longint; virtual; abstract;`	Абстрактный класс, перекрываемый в наследниках. Считывает из потока `Count` байтов в буфер `Buffer`. Возвращает число скопированных байтов
`procedure ReadBuffer(var Buffer; Count: Longint);`	Считывает из потока `Count` байтов в буфер `Buffer`. Возвращает число скопированных байтов
`function Seek(Offset: Longint; Origin: Word): Longint; virtual; abstract;`	Абстрактный класс, перекрываемый в наследниках. Смещает текущую позицию в реальном носителе данных на `Offset` байтов в зависимости от условия `Origin`
`function Write(const Buffer; Count: Longint): Longint; virtual; abstract;`	Абстрактный класс, перекрываемый в наследниках. Записывает в поток `Count` байтов из буфера `Buffer`. Возвращает число скопированных байтов
`procedure WriteBuffer(const Buffer; Count: Longint);`	Записывает в поток `Count` байтов из буфера `Buffer`. Возвращает число скопированных байтов
`function ReadComponent(Instance: TComponent): TComponent;`	Передает данные из потока в компонент `Instance`, заполняя его свойства значениями
`function ReadComponentRes(Instance: TComponent): TComponent;`	Считывает заголовок ресурса компонента `Instance` и значения его свойств из потока
`procedure ReadResHeader;`	Считывает заголовок ресурса компонента из потока
`procedure WriteComponent(Instance: TComponent);`	Передает в поток значения свойств компонента `Instance`
`procedure WriteComponentRes(const ResName: string; Instance: TComponent);`	Записывает в поток заголовок ресурса компонента `Instance` и значения его свойств

Итак, в основе операций считывания и записи данных в потоке лежат методы `Read` и `Write`. Именно они вызываются для реального выполнения операции внутри методов `ReadBuffer` и `WriteBuffer`, `ReadComponent` и `WriteComponent`. Так как класс `TStream` является абстрактным, то методы `Read` и `Write` также абстрактные. В классах-наследниках они перекрываются, обеспечивая работу с конкретным физическим носителем данных.

Метод `Seek` используется для изменения текущей позиции в потоке. "Точка отсчета" позиции зависит от значения параметра `Origin`:

- `soFromBeginning` — смещение должно быть положительным и отсчитывается от начала потока;
- `soFromCurrent` — смещение относительно текущей позиции в потоке;
- `soFromEnd` — смещение должно быть отрицательным и отсчитывается от конца потока.

Группа методов обеспечивает чтение и запись из потока ресурса компонента. Они необходимы при создании компонента на основе данных о нем, сохраненных в формате файлов ресурсов. Для чтения ресурса предусмотрен метод `ReadComponentRes`, в котором последовательно вызываются:

- метод `ReadResHeader` для считывания заголовка ресурса компонента из потока;
- метод `ReadComponent` для считывания значений свойств компонента.

Для записи ресурса в поток применяется метод `WriteComponentRes`.

Класс `THandleStream` инкапсулирует поток, связанный с физическим носителем данных через дескриптор.

Создать поток можно конструктором

```
constructor Create(AHandle: Integer);
```

в параметре которого передается дескриптор. Впоследствии доступ к дескриптору осуществляется через свойство

```
property Handle: Integer;
```

Класс *TFileStream*

Класс `TFileStream` позволяет создать поток для работы с файлами. При этом поток работает с файлом без учета типа хранящихся в нем данных.

Полное имя файла задается в параметре `FileName` при создании потока:

```
constructor Create(const FileName: string; Mode: Word);
```

Параметр `Mode` определяет режим работы с файлом. Он составляется из флагов режима открытия:

- `fmCreate` — файл создается;
- `fmOpenRead` — файл открывается для чтения;
- `fmOpenWrite` — файл открывается для записи;
- `fmOpenReadWrite` — файл открывается для чтения и записи

и флагов режима совместного использования:

- `fmShareExclusive` — файл недоступен для открытия другими приложениями;
- `fmShareDenyWrite` — другие приложения могут читать данные из файла;
- `fmShareDenyRead` — другие приложения могут писать данные в файл;
- `fmShareDenyNone` — другие приложения могут производить с файлом любые операции.

Для чтения и записи из потока есть методы `Read` и `Write`, унаследованные от класса `THandleStream` (листинг 11.5).

Листинг 11.5. Пример работы с потоками

```
procedure TForm1.CopyBtnClick(Sender: TObject);
var Stream1, Stream2: TFileStream;
    IntBuf: array[0..9] of Integer;
begin
 if Not OpenDlg.Execute then Exit;
 try
  Stream1 := TFileStream.Create(OpenDlg.FileName, fmOpenRead);
  Stream1.ReadBuffer(IntBuf, SizeOf(IntBuf));
  try
   Stream2 := TFileStream.Create('TextFile.tmp', fmOpenWrite);
   Stream2.Seek(0, soFromEnd);
   Stream2.WriteBuffer(IntBuf, SizeOf(IntBuf));
  finally
   Stream2.Free;
  end;
 finally
  Stream1.Free;
 end;
end;
```

Обратите внимание, что в данном фрагменте кода функция `Seek` осуществляет запись данных в конец файлового потока.

При необходимости копирования одного файла в другой целиком применяется метод `CopyFrom`, унаследованный от `TStream` (листинг 11.6).

Листинг 11.6. Пример копирования одного файла в другой целиком

```
procedure TForm1.CopyBtnClick(Sender: TObject);
var Stream1, Stream2: TFileStream;
begin
 if Not OpenDlg.Execute then Exit;
 try
  Stream1 := TFileStream.Create(OpenDlg.FileName, fmOpenRead);
  Stream2 := TFileStream.Create('Sample.tmp', fmOpenWrite);
  Stream2.Seek(0, soFromEnd);
  Stream2.CopyFrom(Stream1, Stream1.Size);
 finally
  Stream1.Free;
  Stream2.Free;
 end;
end;
```

Обратите внимание, что в данном случае для определения размера передаваемого потока необходимо свойство `Stream.Size`, которое выдает реальный объем данных, содержащихся в потоке. Функция `SizeOf(Stream)` в этом случае выдаст размер объекта потока, и не более того.

Класс *TMemoryStream*

Класс `TMemoryStream` обеспечивает сохранение данных в адресном пространстве. При этом методы доступа к этим данным остаются теми же, что и при работе с файловыми потоками. Это позволяет использовать адресное пространство для хранения промежуточных результатов работы приложения, а также при помощи стандартных методов осуществлять обмен данными между памятью и другими физическими носителями.

Свойство

```
property Memory: Pointer;
```

определяет область памяти, отведенную для хранения данных потока.

Изменение размера отведенной памяти осуществляется методом

```
procedure SetSize(NewSize: Longint); override;
```

Для очистки памяти потока предусмотрен метод

```
procedure Clear;
```

Чтение/запись данных в память выполняется привычными методами `Read` и `Write`.

Также запись данных в память может осуществляться методами:

- `procedure LoadFromFile(const FileName: string);` — из файла;
- `procedure LoadFromStream(Stream: TStream);` — из другого потока.

Дополнительно имеются методы записи данных в файл или поток:

- `procedure SaveToFile(const FileName: string);`
- `procedure SaveToStream(Stream: TStream);`

Класс *TStringStream*

Так как строковые константы и переменные широко применяются при разработке приложений, то для удобства работы с ними создан специальный класс `TStringStream`. Он обеспечивает хранение строки и доступ к ней во время выполнения приложения.

Он обладает стандартным для потоков набором свойств и методов, добавляя к ним несколько упрощающих работу со строками.

Свойство только для чтения

```
property DataString: string;
```

обеспечивает доступ к хранимой строке.

Методы

```
function Read(var Buffer; Count: Longint): Longint; override;
```

и

```
function Write(const Buffer; Count: Longint): Longint; override;
```

реализуют обычный для потоков способ чтения и записи строки для произвольной переменной `Buffer`.

Метод

```
function ReadString(Count: Longint): string;
```

обеспечивает чтение `Count` байтов строки потока, начиная с текущей позиции.

Метод

```
procedure WriteString(const AString: string);
```

дописывает к строке строку `AString`, начиная с текущей позиции.

При работе с файлами и потоками необходимы дополнительные классы исключительных ситуаций.

Класс `EFCreateError` возникает при ошибке создания файла, а класс `EFOpenError` — при открытии файла.

При чтении/записи данных в поток могут возникнуть ИС `EReadError` и `EWriteError`.

Оповещение об изменениях в файловой системе

Многие программисты задавались вопросом: как получить сигнал от операционной системы о том, что в файловой системе произошли какие-то изменения? Такой вид оповещения позаимствован из ОС UNIX и теперь доступен программистам, работающим с Win32.

Для организации мониторинга файловой системы нужны три функции: `FindFirstChangeNotification`, `FindNextChangeNotification` и `FindCloseChangeNotification`. Первая из них возвращает дескриптор объекта файлового оповещения, который можно передать в функцию ожидания. Объект активизируется тогда, когда в заданной папке произошли те или иные изменения (создание или уничтожение файла или папки, изменение прав доступа и т. д.). Вторая — готовит объект к реакции на следующее изменение. Наконец, с помощью третьей функции следует закрыть ставший ненужным объект.

Листинг 11.7 иллюстрирует, как может выглядеть код метода `Execute` потока, созданного для мониторинга.

Листинг 11.7. Пример мониторинга файловой системы

```
var DirName : string;
...
procedure TSimpleThread.Execute;
var r: Cardinal;
  fn : THandle;
begin
 fn := FindFirstChangeNotifica-
tion(pChar(DirName),True,FILE_NOTIFY_CHANGE_FILE_NAME);
 repeat
 r := WaitForSingleObject(fn,2000);
 if r = WAIT_OBJECT_0 then
  Form1.UpdateList;
 if not FindNextChangeNotification(fn) then break;
 until Terminated;
 FindCloseChangeNotification(fn);
end;
```

На главной форме должны находиться компоненты, нужные для выбора обследуемой папки, а также компонент `TListBox`, в который будут записываться имена файлов (листинг 11.8).

Листинг 11.8. Главная форма приложения для мониторинга файловой системы

```
procedure TForm1.Button1Click(Sender: TObject);
var dir : string;
begin
 if SelectDirectory(dir,[],0) then
  begin
   Edit1.Text := dir;
   DirName := dir;
  end;
end;
procedure TForm1.UpdateList;
var SearchRec: TSearchRec;
begin
  ListBox1.Clear;
  FindFirst(Edit1.Text+'\*.*', faAnyFile, SearchRec);
  repeat
   ListBox1.Items.Add(SearchRec.Name);
  until FindNext(SearchRec) <> 0;
  FindClose(SearchRec);
end;
```

Приложение готово. Чтобы оно стало полнофункциональным, предусмотрите в нем механизм перезапуска потока при изменении обследуемой папки.

Использование отображаемых файлов

Последний, самый нетрадиционный, вид работы с файлами — это так называемые отображаемые файлы.

Вообще говоря, в 32-разрядной Windows под "памятью" подразумевается не только оперативная память (ОЗУ), но также и память, резервируемая операционной системой на жестком диске (виртуальная память). Код и данные отображаются на жесткий диск посредством страничной системы (paging system) подкачки. Страничная система использует для отображения стра-

ничный файл. Необходимый фрагмент виртуальной памяти переносится из страничного файла в ОЗУ и, таким образом, становится доступным.

А что, если так же поступить и с любым другим файлом и сделать его частью адресного пространства? В Win32 это возможно. Для выделения фрагмента памяти должен быть создан специальный системный объект Win32, называемый отображаемым файлом. Этот объект "знает", как соотнести файл, находящийся на жестком диске, с памятью, адресуемой процессами.

Одно или более приложений могут открыть отображаемый файл и получить тем самым доступ к данным этого объекта. Таким образом, данные, помещенные в страничный файл одним приложением, могут быть доступны другим приложениям, если они открыли и используют один и тот же отображаемый файл.

Реализация объектов файлового отображения осуществляется через функции Windows API:

- `CreateFileMapping`
- `MapViewOfFile`
- `UnMapViewOfFile`

Отображаемый файл создается операционной системой при вызове функции `CreateFileMapping`. Этот объект поддерживает соответствие между содержимым файла и адресным пространством процесса, использующего этот файл. Функция `CreateFileMapping` имеет шесть параметров:

```
function CreateFileMapping(hFile: THandle; lpFileMappingAttributes:
PSecurityAttributes; flProtect, dwMaximumSizeHigh, dwMaximumSizeLow:
DWORD; lpName: PChar): THandle;
```

Первый параметр имеет тип `THandle`. Он должен соответствовать дескриптору уже открытого при помощи функции `CreateFile` файла. Если значение параметра `hFile` равно `$FFFFFFFF`, то это приводит к связыванию объекта файлового отображения со страничным файлом операционной системы.

Второй параметр — указатель на запись типа `TSecurityAttributes`. При отсутствии требований к защите данных в Windows NT значение этого параметра всегда равно `nil`. Третий параметр имеет тип `DWORD`. Он определяет атрибут защиты. Если при помощи отображаемого файла вы планируете совместное использование данных, третьему параметру следует присвоить значение `PAGE_READWRITE`.

Четвертый и пятый параметры также имеют тип `DWORD`. Когда выполняется функция `CreateFileMapping`, значение типа `DWORD` четвертого параметра сдвигается влево на четыре байта и затем объединяется со значением пятого параметра посредством операции `and`. Проще говоря, значения объединяются в одно 64-разрядное число, равное объему памяти, выделяемой объекту файлового отображения из страничного файла операционной системы. По-

скольку вы вряд ли попытаетесь осуществить выделение более чем четырех гигабайт данных, то значение четвертого параметра всегда должно быть равно нулю. Используемый затем пятый параметр должен показывать, сколько памяти в байтах необходимо зарезервировать в качестве совместной. Если вы хотите отобразить весь файл, и четвертый и пятый параметры должны быть равны нулю.

Шестой параметр имеет тип `PChar` и представляет собой имя объекта файлового отображения.

Функция `CreateFileMapping` возвращает значение типа `THandle`. В случае успешного завершения возвращаемое функцией значение представляет собой дескриптор созданного объекта файлового отображения. В случае возникновения какой-либо ошибки возвращаемое значение будет равно 0.

Следующая задача — спроецировать данные файла в адресное пространство нашего процесса. Этой цели служит функция `MapViewOfFile`. Функция `MapViewOfFile` имеет пять параметров:

```
function MapViewOfFile(hFileMappingObject: THandle; dwDesiredAccess:
DWORD; dwFileOffsetHigh, dwFileOffsetLow, dwNumberOfBytesToMap: DWORD):
Pointer;
```

Первый параметр имеет тип `THandle`. Его значением должен быть дескриптор созданного объекта файлового отображения — тот, который возвращает функция `CreateFileMapping`. Второй параметр определяет режим доступа к файлу: `FILE_MAP_WRITE`, `FILE_MAP_READ` или `FILE_MAP_ALL_ACCESS`.

Третий и четвертый параметры также имеют тип `DWORD`. Это смещение отображаемого участка относительно начала файла в байтах. В нашем случае эти параметры должны быть установлены в ноль, поскольку значение, которое мы даем пятому (последнему) параметру функции `MapViewOfFile`, также равно нулю.

Пятый и последний параметр функции `MapViewOfFile`, как и предыдущие параметры, имеет тип `DWORD`. Он предназначен для определения (в байтах) количества данных объекта файлового отображения, которые надо отобразить в процесс (сделать доступными для вас). Для достижения наших целей это значение должно быть установлено в ноль, что означает автоматическое отображение в процесс всех данных, выделенных перед этим функцией `CreateFileMapping`.

Значение, возвращаемое функцией `MapViewOfFile`, имеет тип "указатель". Если функция отработала успешно, то она вернет начальный адрес данных объекта файлового отображения.

Следующий фрагмент кода (листинг 11.9) демонстрирует вызов функции `MapViewOfFile`.

Листинг 11.9. Работа с функцией `MapViewOfFile`

```
var
  hMappedFile: THandle;
  pSharedBuf:  PChar;
begin
 hMappedFile :=
CreateFileMapping(FHandle, nil, PAGE_READWRITE, 0, 0, 'SharedBlock');
 if (hMappedFile = 0) then
    ShowMessage('Mapping error!')
    else
    begin
      pSharedBuf :=
      MapViewOfFile(hMappedFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
      if (pSharedBuf = nil) then
         ShowMessage ('MapView error');
    end;
end;
```

После того как получен указатель `pSharedBuf`, вы можете работать со своим файлом как с обычной областью памяти, не заботясь о вводе, выводе, позиционировании и т. п. Все эти проблемы берет на себя файловая система.

Последние две функции, имеющие отношение к объекту файлового отображения, называются `UnMapViewOfFile` и `CloseHandle`. Функция `UnMapViewOfFile` делает то, что и подразумевает ее название. Она прекращает отображение в адресное пространство процесса того файла, который перед этим был отображен при помощи функции `MapViewOfFile`. Функция `CloseHandle` закрывает дескриптор объекта файлового отображения, возвращаемый функцией `CreateFileMapping`.

Функция `UnMapViewOfFile` должна вызываться перед функцией `CloseHandle`. Функции `UnMapViewOfFile` передается единственный параметр типа указатель:

```
procedure TClientForm.FormDestroy(Sender: TObject);
begin
 UnMapViewOfFile(pSharedBuf);
 CloseHandle(hFileMapObj);
end;
```

Отображаемые файлы будут встречаться и в других главах этой книги. Неудивительно, ведь это очень мощный инструмент: помимо возможности со-

вместного доступа он позволяет заметно ускорить доступ к файлам, особенно большого размера.

Резюме

При разработке приложений очень часто приходится решать задачи обмена данными между приложениями или приложением и устройством ввода-вывода. При этом большая часть созданного кода обеспечивает работу приложения с файлами.

В этой главе рассмотрены методы, обеспечивающие взаимодействие программы с файловой системой, и примеры их использования.

Для организации обмена данными в приложениях имеются специальные объекты — потоки, которые не только хранят информацию во время выполнения приложения, но и предоставляют разработчику набор стандартных свойств и методов для управления данными.

Трудно однозначно сказать, при изучении каких глав будет полезен материал этой главы. Скорее всего, он понадобится везде.

Глава 12

Использование графики

"Одно изображение стоит тысячи слов", — говорил древнекитайский император Сун. Эти слова верны и в наши времена. 80% информации мозг человека получает по зрительному каналу, и неудивительно, что программисты стараются придать внешнему виду своих программ максимум привлекательности.

Поэтому в Delphi с самого начала появились развитые средства для работы с графическими возможностями Windows, которые можно использовать и для приложений VCL. Этому и посвящена данная глава.

Графические инструменты Delphi

Разработчики Delphi уделили большое внимание возможностям работы с деловой графикой: простота и удобство ее использования напрямую сказываются на простоте и удобстве созданных приложений. Вместо дебрей графического интерфейса Windows разработчик получил несколько инструментов, сколь понятных, столь же и мощных.

В стандартном графическом интерфейсе Windows (GDI) основой для рисования служит дескриптор контекста устройства `hDC` и связанные с ним шрифт, перо и кисть. В состав VCL входят объектно-ориентированные надстройки над последними, назначение которых — удобный доступ к свойствам инструментов и прозрачная для пользователя обработка всех их изменений.

Обязательным для любого объекта, связанного с графикой в Delphi, является событие:

```
property OnChange: TNotifyEvent;
```

Его обработчик вызывается всякий раз, когда меняются какие-то характеристики объекта, влияющие на его внешний вид.

Класс *TFont*

Класс инкапсулирует шрифт Windows. В Delphi допускаются только горизонтально расположенные шрифты. В конструкторе объекта по умолчанию принимается шрифт System цветом `clWindowText` и размером 10 пунктов.

Свойства класса приведены в табл. 12.1.

Таблица 12.1. *Свойства класса* `TFont`

Свойство	Описание
`property Handle: HFont;`	Содержит дескриптор шрифта
`property Name: TFontName;`	Содержит имя (начертание) шрифта, например, Arial

`property Style: TFontStyles;`
`TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);`
`TFontStyles = set of TFontStyle;`
	Содержит стиль (особенности начертания) шрифта: соответственно полужирный, курсив, подчеркнутый и перечеркнутый

`property Color: TColor;` `TColor = -(COLOR_ENDCOLORS + 1)..$2FFFFFF;`	Определяет цвет шрифта
`property Charset: TFontCharset` `TFontCharset = 0..255;`	Содержит номер набора символов шрифта. По умолчанию равно 1 (`DEFAULT_CHARSET`). Для вывода символов кириллицы требуется `RUSSIAN_CHARSET`
`property Pitch: TFontPitch ;` `TFontPitch = (fpDefault, fpVariable, fpFixed);`	Определяет способ установки ширины символов шрифта. Значение `fpFixed` соответствует моноширинным шрифтам; `fpVariable` — шрифтам с переменной шириной символа. Установка `fpDefault` означает принятие того способа, который определен начертанием
`property Height: Integer;`	Содержит значение высоты шрифта в пикселах
`property PixelsPerInch: Integer;`	Определяет число точек на дюйм. Первоначально равно числу точек на дюйм в контексте экрана. Программист не должен изменять это свойство, т. к. оно используется системой для приведения изображения на экране и принтере к одному виду
`property Size: Integer;`	Содержит размер шрифта в пунктах (как принято в Windows). Это свойство связано с `Height` соотношением: `Font.Size := -Font.Height*72/ Font.PixelsPerInch`

Установка этих свойств вручную, как правило, не нужна. Если вы хотите изменить шрифт для какого-то компонента, воспользуйтесь компонентом `TFontDialog`. В нем можно поменять свойства, и сразу увидеть получившийся результат на тестовой надписи. Потом выбранный шрифт присваивается свойству `Font` нужного компонента:

```
if FontDialog1.Execute
  then  Edit1.Font := FontDialog1.Font;
```

Примечание

Если вы хотите, не закрывая диалог, увидеть результат применения шрифта на вашем тексте, включите опцию `fdApplyButton` в свойстве `Options` объекта `TFontDialog` и напишите для него обработчик события `OnApply`. При этом в диалоговом окне появится кнопка **Apply**, по нажатию которой (событие `OnApply`) можно изменить параметры шрифта.

Класс *TPen*

Этот класс инкапсулирует свойства пера — инструмента для рисования линий. В конструкторе по умолчанию создается непрерывное (`psSolid`) черное перо шириной в один пиксел. Свойства класса приведены в табл. 12.2.

Таблица 12.2. *Свойства класса* `TPen`

Свойство	Описание
`property Handle: HPen;`	Содержит дескриптор пера
`property Color: TColor ;`	Определяет цвет пера
`property Mode: TPenMode ;` `TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy, pmMergePenNot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge, pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor);`	Содержит идентификатор одной из растровых операций, которые определяют взаимодействие пера с поверхностью. Эти операции соответствуют стандартным, определенным в Windows
`property Style: TPenStyle ;` `TPenStyle = (psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear, psInsideFrame);`	Определяет стиль линии, рисуемой пером. Соответствующие стили также определены в Windows
`property Width: Integer;`	Содержит значение толщины пера в пикселах

К сожалению, пунктирные и штрихпунктирные линии (стили `psDash`, `psDot`, `psDashDot` и `psDashDotDot`) могут быть установлены только для ли-

ний единичной толщины. Более толстые линии должны быть сплошными, такое ограничение существует в Windows.

Примечание

Операция `pmNotXor` подходит для рисования перемещающихся линий или фигур, например, при выделении мышью какой-либо области. Если вы два раза нарисуете одну и ту же фигуру таким пером, то после первого раза она появится, после второго — полностью сотрется.

Класс *TBrush*

Этот класс инкапсулирует свойства кисти — инструмента для заливки областей. Когда формируется экземпляр этого класса, первоначально создается белая сплошная (`Style=bsSolid`) кисть. Свойства класса приведены в табл. 12.3.

Таблица 12.3. *Свойства класса* `TBrush`

Свойство	Описание
`property Handle: HBrush;`	Содержит дескриптор кисти
`property Color: TColor ;`	Определяет цвет кисти
`property Style: TBrushStyle ;` `TBrushStyle = (bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross);`	Определяет стиль кисти (фактура закраски)
`property Bitmap: TBitmap;`	Содержит битовую карту, определенную пользователем для закраски поверхностей. Если это свойство определено, то свойства `Color` и `Style` недействительны

Шрифт, перо и кисть не могут использоваться самостоятельно. Они являются составными частями специального класса, который и будет сейчас рассмотрен.

Класс *TCanvas*

Этот класс — сердцевина графической подсистемы Delphi. Он объединяет в себе и "холст" (контекст конкретного устройства GDI), и "рабочие инструменты" (перо, кисть, шрифт) и даже "подмастерьев" (набор функций по рисованию типовых геометрических фигур). Будем называть его канвой.

Канва не является компонентом, но она присутствует в качестве свойства во многих других компонентах, которые должны уметь нарисовать себя и ото-

бразить какую-либо информацию. Читатели, знакомые с графикой Windows, узнают в `TCanvas` объектно-ориентированную надстройку над контекстом устройства Windows (Device Context, DC).

Дескриптор устройства, над которым "построена" канва, может быть востребован для различных низкоуровневых операций. Он задается свойством:

```
property Handle: HDC;
```

Для рисования канва включает в себя шрифт, перо и кисть:

```
property Font: TFont;
property Pen: TPen;
property Brush: TBrush;
```

Кроме того, можно рисовать и поточечно, получив доступ к каждому пикселу. Значение свойства:

```
property Pixels[X, Y: Integer]: TColor;
```

соответствует цвету точки с координатами *X*, *Y*.

Необходимость отрисовывать каждую точку возникает нередко. Однако если нужно модифицировать все или хотя бы многие точки изображения, свойство `Pixels` надо сразу отбросить — настолько оно неэффективно. Гораздо быстрее редактировать изображение при помощи свойства `ScanLine` объекта `TBitmap`, об этом рассказано далее.

Канва содержит методы-надстройки над всеми основными функциями рисования GDI Windows и свойства, которые приведены в табл. 12.4. При их рассмотрении имейте в виду, что все геометрические фигуры рисуются текущим пером и закрашиваются текущей кистью. Кисть и перо при этом имеют текущий цвет.

Таблица 12.4. *Методы и свойства класса* `TCanvas`

Метод	Описание
`procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);`	Метод рисует сегмент эллипса. Эллипс определяется описывающим прямоугольником (*X*1, *Y*1) — (*X*2, *Y*2); его размеры должны лежать в диапазоне от 2 до 32 767 точек. Начальная точка сегмента лежит на пересечении эллипса и луча, проведенного из его центра через точку (*X*3, *Y*3). Конечная точка сегмента лежит на пересечении эллипса и луча, проведенного из его центра через точку (*X*4, *Y*4). Сегмент рисуется против часовой стрелки

Таблица 12.4 *(продолжение)*

Метод	Описание
`procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);`	Рисует хорду и заливает отсекаемую ею часть эллипса. Начальная и конечная точки определяются как в методе `Arc`
`procedure Ellipse(X1, Y1, X2, Y2: Integer);`	Рисует и закрашивает эллипс, вписанный в прямоугольник (*X*1, *Y*1)—(*X*2, *Y*2)
`procedure LineTo(X, Y: Integer);`	Проводит линию текущим пером из текущей точки в (*X*, *Y*)
`procedure MoveTo (X, Y: Integer);`	Перемещает текущее положение пера (свойство `PenPos`) в точку (*X*, *Y*)
`procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap; const Source: TRect; Color: TColor);`	Выполняет специальное копирование. Прямоугольник `Source` из битовой карты `Bitmap` копируется в прямоугольник `Dest` на канве; при этом цвет `Color` заменяется на цвет текущей кисти (`Brush.Color`) С помощью этого метода можно нарисовать "прозрачную" картинку. Для этого нужно выбрать соответствующий фону цвет кисти и затем заменить на него фоновый или наиболее часто встречающийся цвет битовой карты (см. `Bitmap.TransparentColor`)
`procedure CopyRect(const Dest: TRect; Canvas: TCanvas; const Source: TRect);`	Выполняет копирование прямоугольника `Source` из канвы `Canvas` в прямоугольник `Dest` в области самого объекта
`procedure FillRect(const Rect: TRect);`	Выполняет заливку прямоугольника (текущей кистью)
`procedure FrameRect(const Rect: TRect);`	Выполняет оконтуривание прямоугольника цветом текущей кисти (без заполнения)
`procedure Draw(X, Y: Integer; Graphic: TGraphic);`	Осуществляет рисование графического объекта `Graphic` (точнее, вызов метода его рисования) в области с верхним левым углом (*X*, *Y*)
`procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);`	Осуществляет рисование объекта `Graphic` в заданном прямоугольнике `Rect`. Если их размеры не совпадают, `Graphic` масштабируется
`procedure DrawFocusRect(const Rect: TRect);`	Выполняет отрисовку прямоугольной рамки из точек (как на элементе, имеющем фокус ввода). Поскольку метод использует логическую операцию XOR (исключающее ИЛИ), повторный вызов для того же прямоугольника приводит изображение к начальному виду

Таблица 12.4 *(продолжение)*

Метод	Описание
`procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);` `TFillStyle = (fsSurface, fsBorder);`	Производит заливку области текущей кистью. Процесс начинается с точки (*X*, *Y*). Если режим `FillStyle` равен `fsSurface`, то он продолжается до тех пор, пока есть соседние точки с цветом `Color`. В режиме `fsBorder` закрашивание, наоборот, прекращается при выходе на границу с цветом `Color`
`procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);`	Рисует сектор эллипса, описываемого прямоугольником (*X*1, *Y*1)—(*X*2, *Y*2). Стороны сектора лежат на лучах, проходящих из центра эллипса через точки (*X*3, *Y*3) и (*X*4, *Y*4)
`procedure Polygon(const Points: array of TPoint);`	Строит многоугольник, используя массив координат точек `Points`. При этом последняя точка соединяется с первой и внутренняя область закрашивается
`procedure Polyline(const Points: array of TPoint);`	Строит ломаную линию, используя массив координат точек `Points`
`procedure PolyBezier(const Points: array of TPoint);`	Строит кривую Безье (кубический сплайн), используя массив координат точек `Points`
`procedure PolyBezierTo(const Points: array of TPoint);`	Строит кривую Безье (кубический сплайн), используя массив координат точек `Points`. Текущая точка является первой
`procedure Rectangle(X1, Y1, X2, Y2: Integer);`	Рисует прямоугольник с верхним левым углом в (*X*1, *Y*1) и нижним правым в (*X*2, *Y*2)
`Procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);`	Рисует прямоугольник с закругленными углами. Координаты вершин — те же, что и в методе `Rectangle`. Закругления рисуются как сегменты эллипса с размерами осей по горизонтали и вертикали *X*3 и *Y*3
`function TextHeight(const Text: string): Integer;`	Задает высоту строки `Text` в пикселах
`function TextWidth(const Text: string): Integer;`	Задает ширину строки `Text` в пикселах
`procedure TextOut(X, Y: Integer; const Text: string);`	Производит вывод строки `Text`. Левый верхний угол помещается в точку канвы (*X*, *Y*)
`procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);`	Производит вывод текста с отсечением. Как и в `TextOut`, строка `Text` выводится с позиции (*X*, *Y*); при этом часть текста, лежащая вне пределов прямоугольника `Rect`, отсекается и не будет видна

Таблица 12.4 *(окончание)*

Свойство	Описание
`property ClipRect: TRect;`	Определяет область отсечения канвы. То, что при рисовании попадает за пределы этого прямоугольника, не будет изображено. Свойство доступно только для чтения — его значение переустанавливается системой в контексте устройства, с которым связана канва
`property PenPos: TPoint;`	Содержит текущую позицию пера канвы (изменяется посредством метода `MoveTo`)

Метод

```
procedure Refresh;
```

сбрасывает текущие шрифт, перо и кисть, заменяя их на стандартные, заимствованные из установок Windows (`BLACK_PEN`, `HOLLOW_BRUSH`, `SYSTEM_FONT`).

Предусмотрено два события для пользовательской реакции на изменение канвы:

```
property OnChange: TNotifyEvent;
property OnChanging: TNotifyEvent;
```

Эти события возникают при изменениях свойств и вызовах методов `TCanvas`, меняющих вид канвы (т. е. при любом рисовании). В методе `MoveTo`, например, они не возникают). Отличие их в том, что событие `OnChanging` вызывается до начала изменений, а `OnChange` — после их завершения.

Идентификатор (код) растровой операции при копировании прямоугольных блоков содержится в свойстве:

```
property CopyMode: TCopyMode;
TCopyMode = Longint;
```

и определяет правило сочетания пикселов, копируемых на канву, с ее текущим содержимым. При этом можно создавать разные изобразительные эффекты. В Delphi определены следующие константы кодов: `cmBlackness`, `cmDstInvert`, `cmMergeCopy`, `cmMergePaint`, `cmNotSrcCopy`, `cmNotSrcErase`, `cmPatCopy`, `cmPatInvert`, `cmPatPaint`, `cmSrcAnd`, `cmSrcCopy`, `cmSrcErase`, `cmSrcInvert`, `cmSrcPaint`, `cmWhiteness`.

Все они стандартно определены в Windows, и подробное их описание можно найти в документации по GDI. Значением свойства `CopyMode` по умолчанию является `cmSrcCopy` — копирование пикселов источника поверх существующих.

Из возможностей, появившихся в классе `TCanvas`, следует отметить поддержку рисования кривых (полиномов) Безье. Эта возможность впервые появилась в API Windows NT. Для построения одной кривой нужны мини-

мум четыре точки: начальная, конечная и две опорные. По ним будет построена кривая второго порядка. Если задан массив точек, они используются для построения последовательных кривых, причем последняя точка одной кривой является первой для следующей кривой.

Хорошей иллюстрацией объекта `TCanvas` может послужить пример `GraphEx`, поставляемый вместе с Delphi (папка ..\Doc\GraphEx). Есть только одно "но" — он приводится в неизменном виде, начиная с версии Delphi 1.0. Поэтому сделаем часть работы за программистов Borland. В нашем примере модернизированы панели инструментов — они выполнены на компонентах `TToolBar` и `TControlBar`; добавлена поддержка файлов JPEG; и, наконец, добавлена возможность рисования кривых Безье. Обновленный внешний вид главной формы примера GraphEx показан на рис. 12.1.

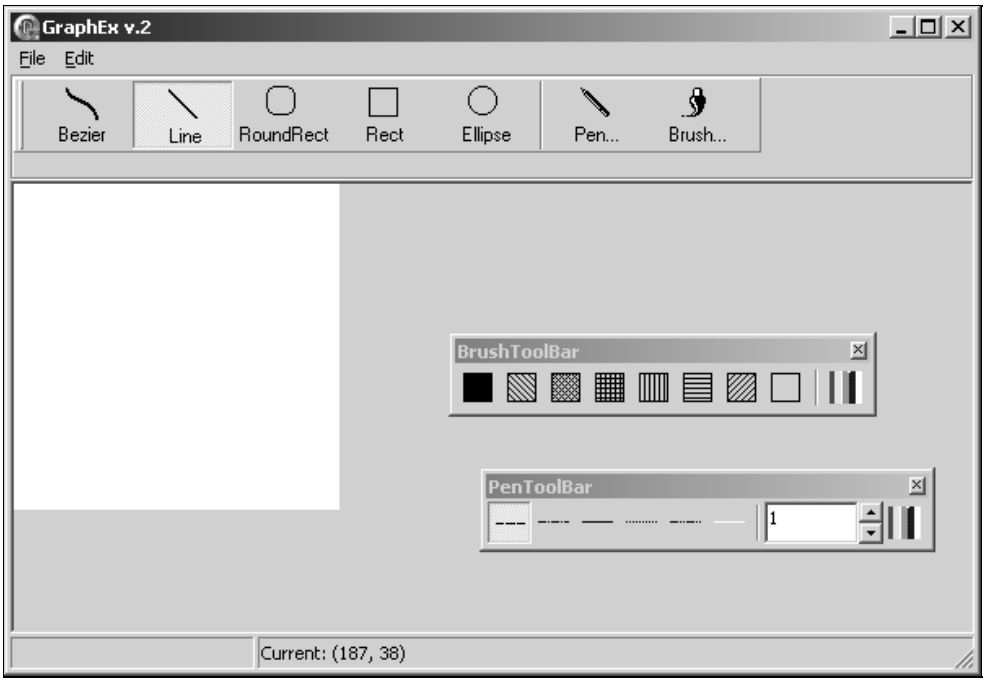


Рис. 12.1. Так теперь выглядит главная форма примера GraphEx

Где же найти ту канву, на которой предстоит рисовать? Во-первых, ею снабжены все потомки классов `TGraphicControl` и `TCustomControl`, т. е. почти все визуальные компоненты из Палитры инструментов, в том числе и форма. Во-вторых, канву имеет растровая картинка (класс `TBitmap`), вы можете писать и рисовать не на пустом месте, а на готовом изображении (об этом в разделе *"Класс TBitmap" данной главы*). Но иногда нужно рисовать и

прямо на экране! В этом случае придется прибегнуть к функциям API. Функция `GetDC` возвращает контекст устройства заданного окна, если ей передается параметр 0 — то всего экрана:

```
ScreenCanvas := TCanvas.Create;
ScreenCanvas.Handle := GetDC(0);
// рисование на ScreenCanvas
ReleaseDC(0, ScreenCanvas.Handle);
ScreenCanvas.Free;
```

Необходимость рисования на экране возникает, например, в программах сохранения экрана (Screen savers).

Когда и где следует рисовать? Этот вопрос далеко не риторический, как может показаться с первого взгляда.

Помимо графических примитивов, таких как линии и фигуры, на канве можно разместить готовые изображения. Для их описания создан класс `TGraphic`.

Класс *TGraphic*

Канва, перо, кисть и шрифт нужны, чтобы нарисовать свою картинку. Чтобы загрузить готовую, необходимы объекты, "понимающие" графические форматы Windows.

Абстрактный класс `TGraphic` является родительским для трех видов изображений, общепринятых в графике Windows: значка (компонент `TIcon`), метафайла (компонент `TMetafile`) и растровой картинки (компонент `TBitmap`). Четвертым потомком `TGraphic` является `TJPEGImage` — сжатая растровая картинка в формате JPEG.

Работая над приложением в Delphi, вы никогда не будете создавать объекты класса `TGraphic;`, но переменной этого типа можно присваивать указатель на любой из перечисленных классов-потомков.

Метод:

```
procedure Assign(Source: TPersistent);
```

переопределяет одноименный метод предка, допуская полиморфное присваивание графических объектов.

Загрузку и выгрузку графики в поток осуществляют методы:

```
procedure LoadFromStream(Stream: TStream);
procedure SaveToStream(Stream: TStream);
```

а загрузку и выгрузку в файл — методы:

```
procedure LoadFromFile(const Filename: string);
procedure SaveToFile(const Filename: string);
```

Эти методы создают соответствующий файловый поток и затем вызывают методы LoadFromStream/SaveToStream.

Два метода осуществляют взаимодействие с буфером обмена Windows:

```
procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle;
APalette: HPALETTE);
```

```
procedure SaveToClipboardFormat(var AFormat: Word; var AData: THandle;
var APalette: HPALETTE);
```

Здесь AFormat — выбранный графический формат; AData и APalette — данные и палитра (если она требуется). Потомок должен иметь свой формат представления в буфере обмена и уметь обрабатывать данные, представленные в нем.

Загрузка больших графических файлов может продолжаться очень долго. Чтобы скрасить пользователю ожидание, программист может обработать событие OnProgress:

```
type
```

```
TProgressStage = (psStarting, psRunning, psEnding);
```

```
TProgressEvent = procedure (Sender: TObject; Stage: TProgressStage;
PercentDone: Byte; RedrawNow: Boolean; const
```

```
 R: TRect; const Msg: string) of object;
```

```
property OnProgress: TProgressEvent;
```

Оно вызывается графическими объектами во время длительных операций. Параметр Stage означает стадию процесса (начало/протекание/завершение), а PercentDone — процент выполненной работы. Сразу оговоримся, что не все из тех объектов, которые будут нами описаны, вызывают обработчик события OnProgress.

Свойство:

```
property Empty: Boolean;
```

устанавливается в True, если графический объект пуст (в него не загружены данные).

Высота и ширина графического объекта задаются свойствами:

```
property Height: Integer;
```

```
property Width: Integer;
```

Для каждого дочернего типа эти параметры вычисляются своим способом. Наконец, свойство:

```
property Modified: Boolean;
```

показывает, модифицировался ли данный графический объект. Это свойство устанавливается в значение True внутри обработчика события OnChange.

Многие графические объекты при отрисовке должны быть прозрачными. Некоторые из них прозрачны всегда (значок, метафайл), некоторые — в зависимости от значения свойства

```
property Transparent: Boolean;
```

Класс *TPicture*

Этот класс-надстройка над `TGraphic`, точнее — над его потомками. Он содержит поле `Graphic`, которое может включать объекты классов `TBitmap`, `TIcon`, `TMetafile` и `TJPEGImage`. Предназначение `TPicture` — управлять вызовами соответствующих методов, скрывая при этом хлопоты с определением типа графического объекта и детали его реализации.

Кроме того, на уровне `TPicture` определены возможности регистрации и использования других — определенных пользователем — классов графических объектов, порожденных от `TGraphic`. Доступ к графическому объекту осуществляется посредством свойства:

```
property Graphic: TGraphic;
```

Если графический объект имеет один из трех предопределенных типов, то к нему можно обратиться и как к одному из свойств:

- `property Bitmap: TBitmap`
- `property Icon: TIcon`
- `property Metafile: TMetafile`

Обращаясь к этим функциям, нужно быть осторожным. Если в поле `Graphic` хранился объект одного класса, а затребован — другого, то прежний объект уничтожается, а вместо него создается пустой объект требуемого класса:

```
Image1.Picture.LoadFromFile('myicon.ico'); // Создан и загружен
                                           // объект класса TIcon
MyBitmap := Image1.Picture.Bitmap; // прежний TIcon уничтожается
```

Если же вы описали свой класс (допустим, `TGIFImage`), то к его методам и свойствам следует обращаться так:

```
(Graphic as TGIFImage).MyProperty := MyValue;
```

Перечислим остальные методы и свойства.

- `procedure LoadFromFile(const Filename: string);` — анализирует расширение имени файла `FileName` и если оно известно (зарегистрировано), то создается объект нужного класса и вызывается его метод `LoadFromFile`. В противном случае возникает исключительная ситуация `EInvalidGraphic`. Стандартными расширениями являются ICO,

WMF (EMF) и BMP. Если подключить к приложению модуль JPEG.PAS, то можно будет загрузить и файлы с расширениями JPG и JPEG.

- `procedure SaveToFile(const Filename: string);` — сохраняет графику в файле, вызывая соответствующий метод объекта `Graphic`.
- `procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle; APalette: HPALETTE);` — во многом аналогичен методу `LoadFromFile`. Если формат `AFormat` найден среди зарегистрированных, то `AData` и `APalette` передаются для загрузки методу соответствующего объекта. Изначально зарегистрированных форматов три: битовое изображение CF_BITMAP и метафайлы CF_METAFILEPICT и CF_ENHMETAFILE.
- `procedure SaveToClipboardFormat(var AFormat: Word; var AData: THandle; var APalette: HPALETTE);` — сохраняет графику в буфере обмена, вызывая метод объекта `Graphic`.
- `procedure Assign(Source: TPersistent);` — переписан таким образом, чтобы присваиваемый объект мог принадлежать как классу `TPicture`, так и `TGraphic` или любого его потомка. Кроме того, он может быть равен `nil` — в этом случае поле `Graphic` очищается с удалением прежнего объекта.
- `class function SupportsClipboardFormat(AFormat: Word): Boolean;` — возвращает `True`, если формат `AFormat` поддерживается классом `TPicture` (зарегистрирован в системе). Напомним, что методы класса можно вызывать через ссылку на класс, без создания экземпляра объекта.

Методы класса

```
class procedure RegisterFileFormat(const AExtension, ADescription:
string; AGraphicClass: TGraphicClass);
```

```
class procedure RegisterClipboardFormat(AFormat: Word; AGraphicClass:
TGraphicClass);
```

предназначены для создателей новых графических классов. Они позволяют зарегистрировать формат файла и буфера обмена соответственно и связать их с созданным классом-потомком `TGraphic`, который умеет читать и записывать информацию в этом формате.

Ширина и высота картинки определяются свойствами:

```
property Width: Integer;
```

```
property Height: Integer;
```

Значения этих свойств всегда те же, что и у объекта из свойства `Graphic`.

Все три разновидности графических объектов имеют свои системы кэширования. Это означает, что на один реально существующий в системе (и занимающий долю ресурсов!) дескриптор могут одновременно ссылаться

несколько объектов. Реализуется такое связывание через метод `Assign`. Выражение:

```
Icon1.Assign(Icon2);
```

означает, что два этих объекта разделяют теперь один фактически находящийся в памяти значок.

Более простым является кэширование для `TIcon` и `TMetafile`, которые умеют только отображать себя и не предназначены для редактирования (создатели Delphi считают, что дескриптор графического объекта дается программисту не для того, чтобы "ковыряться" в нем на уровне двоичных кодов). Гораздо сложнее устроен механизм кэширования для канала `TBitmap`, который имеет свою канву для рисования.

Внутреннее представление информации в графических объектах двоякое: она может храниться как поток типа `TMemoryStream` (в него загружается содержимое соответствующего файла), как область памяти с дескриптором (структура которой зависит от типа графического объекта) и одновременно в двух этих видах, содержимое которых автоматически синхронизируется. Поэтому будьте готовы, что загрузка изображения потребует вдвое большего объема памяти (это особенно актуально для больших картинок).

Кого-то может удивить отсутствие объявленных методов рисования, похожих на метод `Draw` для классов `TIcon`, `TMetafile` и `TBitmap`. Объяснение простое — в процессе рисования они играют пассивную роль: рисуют не они, рисуют их. Все рисование должно осуществляться через вызовы методов `Draw` и `StretchDraw` канвы, содержащей графику, так как канва соответствует тому контексту, в котором должна осуществляться операция.

Рассмотрим предопределенные графические классы.

Класс *TMetafile*

Инкапсулирует свойства метафайла Windows. Метафайл может быть стандартного (формат WMF) и расширенного (формат EMF) формата, обладающего расширенными возможностями. Соответственно в объекте `TMetafile` имеется свойство

```
property Enhanced: Boolean;
```

Внутреннее представление метафайла всегда новое (EMF), и устанавливать свойство `Enhanced` в `False` следует только для обеспечения совместимости со старыми программами.

В классе `TMetafile` перекрываются методы `Assign`, `LoadFromStream`, `SaveToStream`, `LoadFromClipboardFormat`, `SaveToClipboardFormat`. В буфер обмена объект помещает свое содержимое в формате `CF_ENHMETAFILE`.

Помимо общих, класс имеет следующие свойства:

- дескриптор метафайла `property Handle: HMETAFILE;`
- свойство `property Inch: Word`. Число точек на дюйм в координатной системе метафайла. Связано с установленным режимом отображения.
- свойства `property MMHeight: Integer;`

  `property MMWidth: Integer;`

  настоящие высота и ширина метафайла в единицах, равных 0,01 мм. Свойства `Height` и `Width` задаются в пикселах.
- в метафайл можно добавить свою палитру:

  `property Palette: HPalette;`
- вы можете увековечить себя, установив два свойства метафайла:

  `property Description: string;`

  `property CreatedBy: string;`

  Содержащаяся в них информация записывается в файл и может быть прочитана благодарными потомками.

Класс *TIcon*

Этот класс инкапсулирует значок Windows.

Не пытайтесь изменить размеры значка, они по определению постоянны (и равны `GetSystemMetrics(SM_CXICON)` и `GetSystemMetrics(SM_CYICON)`). При попытке присвоить новые значения возникает исключительная ситуация `EInvalidGraphicOperation`. Значок нельзя также читать и писать в буфер обмена, так как в Windows нет соответствующего формата.

Свойство `Transparent` для значка всегда равно `True`. Изменить его нельзя, т. к. значки прозрачны по определению.

В этом классе перекрываются методы класса `TGraphic`: `Assign`, `LoadFromStream` и `SaveToStream`.

Дополнительно определены:

- `property Handle: HICON;` — дескриптор значка;
- `function ReleaseHandle: HICON;` — метод "отдает" дескриптор (возвращает его значение, обнуляя ссылку на него в объекте).

Класс *TBitmap*

Класс `TBitmap` является основой растровой графики в Delphi. В первых версиях среды этот класс соответствовал битовой карте, зависимой от устройства (Device Dependent Bitmap, DDB). Этот формат хорош для деловой графи-

ки — отображения небольших картинок с малой глубиной цвета, например, на кнопках. Формат DDB родился во времена первых версий Windows, когда еще не было графических ускорителей и кое-где еще помнили о EGA. Поэтому и форматы хранения были привязаны к определенным видеорежимам.

Со временем аппаратура совершенствовалась, увеличивалось и число поддерживаемых видеорежимов. Появились режимы High Color (15—16 бит на точку) и True Color (24 бита на точку). Все это привело к тому, что картинка стала храниться в аппаратно-независимом формате (Device Independent Bitmap, DIB), а проблемы ее быстрого отображения легли на аппаратуру и драйверы.

За формат битовой карты (DIB или DDB) отвечает свойство:

```
type TBitmapHandleType = (bmDIB, bmDDB);
property HandleType: TBitmapHandleType;
```

По умолчанию устанавливается режим `bmDIB`. Впрочем, можно заставить приложение, написанное на Delphi, вернуться к старому типу. Для этого нужно установить глобальную переменную `DDBsOnly` (пространство имен `Borland.Vcl.Graphics`) в `True`. Впрочем, необходимость этого сомнительна. Все новые видеокарты и драйверы к ним, а также графические интерфейсы (такие как DirectX) оптимизированы для использования DIB.

Желаемую глубину цвета битовой карты можно узнать и переустановить, меняя значение свойства:

```
TPixelFormat = (pfDevice, pf1bit, pf4bit, pf8bit, pf15bit, pf16bit,
pf24bit, pf32bit, pfCustom);
property PixelFormat: TPixelFormat;
```

Формат `pfDevice` соответствует битовой карте DDB. Глубина цвета в 1, 4 и 8 бит на пиксел — традиционная и предусматривает наличие у изображения палитры. Другие режимы обеспечивают хранение непосредственных яркостей точек в каждом из трех основных цветов: красном (R), зеленом (G) и синем (B). Разрядность 15 бит соответствует распределению бит 5-5-5 (RGB555), 16 бит — RGB 565, 24 бита — RGB888. Режим 32 бита похож на 24-битный, но в нем добавлен четвертый канал (альфа-канал), содержащий дополнительную информацию о прозрачности каждой точки. Формат `pfCustom` предназначен для реализации программистом собственных графических конструкций. В стандартном классе `TBitmap` установка свойства `PixelFormat` в `pfCustom` приведет к ошибке, поэтому использовать его нужно только в написанных вами потомках `TBitmap`.

Битовая карта является одним из видов ресурсов. Естественно, что класс `TBitmap` поддерживает загрузку из ресурсов приложения:

```
procedure LoadFromResourceID(Instance: THandle; ResID: Integer);
procedure LoadFromResourceName(Instance: THandle; const ResName: string);
```

Здесь `Instance` — глобальная переменная пространства имен `System`, хранящая уникальный идентификатор запущенной копии приложения (или динамической библиотеки).

Канва битовой карты доступна через свойство:

```
property Canvas: TCanvas;
```

С ее помощью можно рисовать на поверхности растрового изображения. Обратите внимание, что никакие другие потомки `TGraphic` канвы не имеют.

Дескрипторы битовой карты и ее палитры доступны как свойства:

```
property Handle: HBITMAP;
property Palette: HPALETTE;
```

Имея дело с `TBitmap`, учитывайте, что принцип "один объект — один дескриптор" из-за наличия механизма кэширования неверен.

Два метода:

```
function ReleaseHandle: HBITMAP;
function ReleasePalette: HPALETTE;
```

возвращают дескрипторы битовой карты и палитры соответственно и после этого обнуляют дескрипторы, т. е. как бы "отдают" их пользователю.

При любом внешнем обращении к дескриптору битовой карты и любой попытке рисовать на ее канве разделение одной картинки несколькими объектами прерывается, и объект получает собственную копию содержимого дескриптора. Для этого есть методы:

- `procedure Dormant` — выгружает изображение в поток и уничтожает дескрипторы битовой карты и палитры;
- `procedure FreeImage` — "освобождает" дескриптор битовой карты для дальнейшего использования и внесения изменений. Это означает, что если на данный дескриптор есть ссылки, то он дублируется, а поток очищается.

Битовая карта может быть монохромной и цветной, что определено свойством:

```
property Monochrome: Boolean;
```

Значение `True` соответствует монохромной битовой карте. При его изменении происходит преобразование содержимого к требуемому виду.

За прозрачность битовой карты отвечают свойства:

- `property TransparentColor: TColor;`
- `type TTransparentMode = (tmAuto, tmFixed);`
- `property TransparentMode: TTransparentMode;`

Если свойство TransparentMode установлено в tmAuto, за прозрачный (фоновый) принимается цвет верхнего левого пиксела. В противном случае этот цвет берется из свойства TransparentColor.

Битовая карта может являться маской для других битовых карт. В этом случае она превращается в двухцветную, где в белый цвет превращаются точки фона (см. TransparentColor), а в черный — все остальные. Для поддержки этого режима служат следующие методы и свойства:

- procedure Mask(TransparentColor: TColor);
- property MaskHandle: HBitmap;
- function ReleaseMaskHandle: HBitmap;

Наконец, последним по счету будет рассмотрено очень важное свойство битовой карты — TBitmap. Если формат ее хранения — DIB, то можно получить доступ к данным самой битовой карты:

```
property ScanLine[Row: Integer]: IntPtr; Pointer;
```

Это свойство представляет собой массив указателей на строки с данными битовой карты. Параметр Row содержит номер строки. Следует помнить, что в большинстве случаев строки в битовой карте упорядочены в памяти снизу вверх и фактически первой после заголовка хранится нижняя строка. Код, возвращающий значение свойства ScanLine, это учитывает, поэтому не удивляйтесь, если с ростом параметра Row значение свойства уменьшается.

Внутри строки данные упорядочены в соответствии с форматом (PixelFormat). Для формата pf8bit все просто — каждый байт в строке соответствует одному пикселу. Для pf15bit и pf16bit пикселу соответствуют два байта (в этих 16 битах упакованы данные о трех каналах), pf24bit — три байта (по байту на канал).

Листинг 12.1 иллюстрирует, как может выглядеть обработчик события OnMouseMove, выводящий на панель состояния информацию о яркости в данной точке (подразумевается, что формат битовой карты 8 или 24 бита).

Листинг 12.1. Пример обработчика события OnMouseMove

```
procedure TMainForm.Image1MouseMove(Sender: TObject; Shift: TShiftState;
X, Y: Integer);
begin
  if not Assigned(Image1.Picture.Bitmap) then Exit;
with Image1.Picture.Bitmap. do
case PixelFormat of
 pf8bit: Statusbar1.SimpleText := Format('x: %d y: %d b: %d',[x, y,
pByteArray(ScanLine[y])^[x] ]);
```

```
 pf24bit: Statusbar1.SimpleText := Format('x: %d y: %d R: %d,G: %d, B: %d',
 [x,y, pByteArray(ScanLine[y])^[3*x], pByteArray(ScanLine[y])^[ 3*x+1],
pByteArray(ScanLine[y])^[ 3*x+2]]);
end;
```

Само значение свойства `ScanLine` изменить нельзя (оно доступно только для чтения). Но можно изменить данные, на которые оно указывает. Вот так можно получить негатив 24-битной картинки:

```
Var line : pByteArray;
...
For i:=0 to Image1.Picture.Bitmap.Height – 1 do
 Begin
 Line := Image1.Picture.Bitmap.ScanLine[i];
 For j:=0 to Image1.Picture.Bitmap.Width * 3 – 1 do
 Line^[j] := 255 – Line^[j];
 End;
```

Если вы хотите решать более серьезные задачи — на уровне профессиональных средств — на помощь может прийти библиотека обработки изображений фирмы Intel — Intel Image Processing Library. Этот набор инструментов позволяет разработчику включать в программы алгоритмы обработки изображений, написанные и оптимизированные специально для процессоров фирмы Intel. Библиотека является свободно распространяемой, и последняя ее версия находится на Web-сайте фирмы. Интерфейсы к функциям библиотеки для Delphi разработаны авторами этой книги и вместе с примерами находятся на прилагаемом к книге диске.

Примечание

В Delphi можно столкнуться с "тезкой" рассматриваемого объекта — структурой `TBitmap`, описанной в файле WINDOWS.PAS. Поскольку обе они относятся к одной и той же предметной области, часто возникают коллизии, приводящие к ошибкам. Напомним, чтобы отличить структуры-синонимы, следует использовать имя модуля, в котором они описаны. Поэтому если в вашей программе есть модули Windows и Graphics, то описывайте и употребляйте типы `Windows.TBitmap` и `Graphics.TBitmap`.

В состав Windows входят карточные игры (точнее, пасьянсы), которые черпают ресурсы из динамической библиотеки cards.dll. Если вы откроете эту библиотеку в редакторе ресурсов, то увидите там изображения всех пятидесяти двух карт и десятка вариантов их рубашек (оборотных сторон). Используем эту возможность для рисования карт. Листинг 12.2 иллюстрирует, как загружается битовая карта для рубашки.

Листинг 12.2. Пример загрузки битовой карты

```
var CardsDll : THandle;
     BackBitmap : Graphics.TBitmap;
initialization
 CardsDll := LoadLibraryEx('cards.dll',0, LOAD_LIBRARY_AS_DATAFILE);
 BackBitmap := Graphics.TBitmap.Create;
 BackBitmap.LoadFromResourceID(CardsDll,64);
finalization
 BackBitmap.Free;
 FreeLibrary(CardsDll);
end.
```

Аналогичным образом можно загрузить битовые карты для всей колоды. При показе карты, в зависимости от того, открыта она или закрыта, отрисовывается один из объектов `TBitmap`:

```
if Known then //карта открыта
   Canvas.StretchDraw(ClientRect, FaceBitmap)
  else
   Canvas.StretchDraw(ClientRect,BackBitmap)
 end;
```

Графический формат JPEG. Класс *TJPEGImage*

В 1988 году был принят первый международный стандарт сжатия неподвижных изображений. Он был назван по имени группы, которая над ним работала, — JPEG (Joint Photographic Expert Group). Дело в том, что стандартные архиваторы (ZIP, ARJ) и традиционные алгоритмы сжатия в форматах GIF, TIFF и PCX не могут достаточно сильно сжать полутоновую или цветную картинку (типа фотографии) — максимум в 2—3 раза. Примененный в JPEG алгоритм позволяет достичь сжатия в десятки раз, правда, при этом изображение подвергается необратимому искажению, и из него пропадает часть деталей. Бессмысленно (и вредно!) подвергать хранению в формате JPEG чертежи, рисунки, а также любые изображения с малым числом градаций — он предназначен именно для изображений фотографического качества.

Поддержка формата JPEG реализована в Delphi посредством класса `TJPEGImage`, который является потомком класса `TGraphic`.

Примечание

Название `TJPEGImage` не совсем удачное, к `TImage` этот класс не имеет ни малейшего отношения. Скорее это "двоюродный брат" класса `TBitmap`.

К такому объекту предъявляются двоякие требования. С одной стороны, он должен поддерживать сжатие данных для записи на диск. С другой, — распакованные данные в формате DIB, чтобы по требованию системы отрисовать их. Поэтому объект класса `TJPEGImage` может хранить оба вида данных, а также производить их взаимные преобразования, т. е. сжатие и распаковку. Для этого в нем предусмотрены следующие методы:

- `procedure Compress;`
- `procedure DIBNeeded;`
- `procedure JPEGNeeded;`

Рекомендуется вызывать метод `DIBNeeded` заранее, перед отрисовкой картинки — это ускорит процесс ее вывода на экран.

Кроме того, полезен метод `Assign`, позволяющий поместить в класс `TJPEGImage` объект `TBitmap` и наоборот:

```
MyJPEGImage.Assign(MyBitmap);
MyBitmap.Assign(MyJPEGImage);
```

При этом происходит преобразование форматов.

Свойства `TJPEGImage` можно условно разделить на две группы: используемые при сжатии и при распаковке.

Важнейшим из свойств, нужных при сжатии, является `CompressionQuality`:

```
type TJPEGQualityRange = 1..100;
property CompressionQuality: TJPEGQualityRange;
```

Оно определяет качество сжимаемого изображения и его размер. При малых значениях этого свойства файлы получаются очень маленькими, но с большими искажениями (напомним, что стандарт JPEG предусматривает сжатие с потерями качества). При значениях, близких к 100, потери незаметны, но и размер файла при этом максимален.

Примечание

Заранее предсказать размер сжатого файла нельзя — разные картинки сжимаются по-разному, даже при одном значении `CompressionQuality`.

По умолчанию значение этого свойства равно 75, что обеспечивает разумный компромисс между размером и качеством.

Кроме `CompressionQuality` на качество отображения может повлиять и свойство

```
type TJPEGPerformance = (jpBestQuality, jpBestSpeed);
property Performance: TJPEGPerformance;
```

Оно нужно только при распаковке и отвечает за способ восстановления цветовой палитры из сжатой информации. На качество записываемого изображения оно никак не влияет.

Как и у `TBitmap`, у `TJPEGImage` есть свойство

```
type TJPEGPixelFormat = (jf24Bit, jf8Bit);
property PixelFormat: TJPEGPixelForm;
```

Для рассматриваемого объекта возможных значений всего два: `jf8bit` и `jf24bit`. По умолчанию задан 24-битный формат. Если информация о цвете не нужна, то можно установить свойство `GrayScale` в `True` — в этом случае изображение будет записано (или распаковано) в полутоновом виде (256 оттенков серого).

Свойства `ProgressiveEncoding` и `ProgressiveDisplay` определяют способ показа изображения при распаковке. Первое из них отвечает за порядок записи в файл сжатых компонентов. Если `ProgressiveEncoding` установлено в `True`, начинает играть роль свойство `ProgressiveDisplay`. От его значения зависит, будет ли показываться изображение по мере распаковки (при значении `True`), либо будет сначала полностью распаковано, а потом показано (при значении `False`).

Чтобы организовать предварительный просмотр значительного числа больших изображений, уместно воспользоваться свойством:

```
type TJPEGScale = (jsFullSize, jsHalf, jsQuarter, jsEighth);
property Scale: TJPEGScale;
```

Искушенные в графике специалисты зададут вопрос: зачем оно? Ведь можно прочитать изображение, а затем уменьшить его масштаб стандартными способами? Представление информации в файлах JPEG таково, что можно достаточно просто извлечь изображение сразу в нужном масштабе. Таким образом достигается двойной выигрыш: на времени распаковки и на времени отображения.

Примечание

Печать растровых изображений может вызвать проблемы при согласовании его размеров с размерами листа принтера и его разрешением. Бóльшую часть из них можно снять, изучив пример, поставляемый с Delphi, — jpegProj. Он находится не в папке \Demos, как обычно, а в папке Help\Examples\Jpeg.

В заключение отметим, что класс `TJPEGImage` не имеет своей канвы для рисования, для этого его нужно преобразовать в `TBitmap`.

Компонент *TImage*

Этот компонент служит надстройкой над классом `TPicture` и замыкает всю иерархию графических объектов VCL. Именно на его поверхности и будут отображаться графические объекты, содержащиеся в свойстве

```
property Picture: TPicture;
```

В качестве канвы компонента (свойство `Canvas`) используется канва объекта из свойства `Picture.Graphic`, но только если поле `Graphic` ссылается на объект класса `TBitmap`. Если это не так, то попытка обращения к свойству вызовет исключительную ситуацию `EInvalidOperation`, т. к. рисовать на метафайле или значке нельзя.

Следующие три свойства определяют, как именно графический объект располагается в клиентской области компонента:

- `property AutoSize: Boolean;`

  Означает, что размеры компонента настраиваются по размерам содержащегося в нем графического объекта. Устанавливать его в `True` нужно перед загрузкой изображения из файла или буфера обмена;

- `property Stretch: Boolean;`

  Если это свойство установлено в `True`, то изображение "натягивается" на клиентскую область, при необходимости уменьшая или увеличивая свои размеры. Если оно установлено в `False`, то играет роль следующее свойство;

- `property Center: Boolean;`

  Если это свойство установлено в `True`, изображение центрируется в пределах клиентской области. В противном случае оно располагается в ее верхнем левом углу.

Несмотря на то, что описанию свойств и методов графических объектов здесь отведено уже довольно много места, работа с ними проста и удобна. Программу для просмотра изображений в среде Delphi можно создать буквально "в три счета".

1. Поместите на форму следующие компоненты: область прокрутки `TScrollBox`, на нее — компонент `TImage` (их верхние левые углы должны совпадать), любую кнопку (например, `TButton`) и диалог открытия файлов `TOpenPictureDialog`.

2. Подключите к главному модулю создаваемого приложения модуль JPEG (в предложении `uses`); свойство `AutoSize` компонента `TImage` установите в `True`.
3. Дважды щелкните мышью на кнопке. В появившемся обработчике события `OnClick` напишите такой код:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
 OpenPictureDialog1.Filter := GraphicFilter(TGraphic);
 if OpenPictureDialog1.Execute
 then Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

Приложение готово. Обратите внимание на роль полиморфизма в методе `LoadFromFile`. По расширению файла определяется его формат, и в зависимости от этого создается нужный графический объект.

Использование диалогов для загрузки и сохранения графических файлов

Для удобства открытия картинок существует пара компонентов-диалогов: `TOpenPictureDialog` и `TSavePictureDialog`.

Список форматов открываемых файлов определяется свойством `Filter`. Можно, как в случае со стандартными диалогами `TOpenDialog` или `TSaveDialog`, сформировать их вручную с помощью редактора свойства `Filter`. Можно поступить проще, воспользовавшись готовыми средствами. Для удобства формирования строк графических фильтров существуют три специальные функции:

- `function GraphicFilter(GraphicClass: TGraphicClass): string;`

  Формирует строку с полным текстом графического фильтра, позволяющего открывать все файлы, форматы которых являются потомками параметра `GraphicClass`. Если в качестве параметра этой функции будет передан класс `TGraphic`, то в строке будут перечислены все форматы:

```
'All
(*.jpg;*.jpeg;*.bmp;*.ico;*.emf;*.wmf)|*.jpg;*.jpeg;*.bmp;*.ico;*.emf;
*.wmf|JPEG Image File (*.jpg)|*.jpg|JPEG Image File
(*.jpeg)|*.jpeg|Bitmaps (*.bmp)|*.bmp|Icons (*.ico)|*.ico|Enhanced
Metafiles (*.emf)|*.emf|Metafiles (*.wmf)|*.wmf'
```

  Формат JPEG появляется в перечне, если в приложении используется модуль с тем же названием — JPEG. В приводимом далее примере возникла необходимость совместить фильтры только для классов `TBitmap` и

TJPEGImage, которые не являются предками друг друга. В этом случае получившиеся строки нужно соединить, использовав символ конкатенации |:

```
S := GraphicFilter(TBitmap)+'|'+GraphicFilter(TJpegImage)
```

- `function GraphicExtension(GraphicClass: TGraphicClass): string;`

  Возвращает расширение файла, формат которого соответствует графическому классу GraphicClass. Так, если передать в качестве параметра класс TBitmap, то функция вернет строку 'BMP';

- `function GraphicFileMask(GraphicClass: TGraphicClass): string;`

  Эта функция возвращает перечень расширений файлов с форматами-потомками GraphicClass, перечисленных через точку с запятой.

Для диалогов предусмотрено несколько событий, которые программист может обработать самостоятельно. Первые три достаточно тривиальные: OnShow, OnCanClose и OnClose. Нужно предостеречь программиста: по чьему-то недосмотру последние два вызываются только в случае нормального завершения диалога (нажатием кнопки **Open** или **Save**), а если завершить диалог нажатием кнопки **Cancel** или крестика на заголовке диалога, управления они не получат.

Другие три события связаны с изменениями, которые осуществляет пользователь во время выбора нужного файла. Они происходят в момент изменения формата открываемого файла (событие OnTypeChange), изменения текущей папки (OnFolderChange) и текущего файла (OnSelectionChange).

Но разработчики диалогов не предусмотрели одну очень нужную возможность. Дело в том, что у разных графических форматов возможны различные опции и варианты. Если вы имеете опыт работы с графическими пакетами вроде Adobe Photoshop или Corel, то знаете, что в зависимости от выбранного формата сохранения данных диалог изменяет свой внешний вид: к нему добавляются элементы управления, соответствующие параметрам формата.

Поступим так и мы, предусмотрев настройку при сохранении файлов формата JPEG. Для этого будет использовано событие OnTypeChange компонента TSavePictureDialog. Для события нужно проверить значение свойства FilterIndex. Если оно больше 1 (т. е. выбраны файлы формата JPEG), нужно увеличить высоту окна диалога и разместить на нем дополнительные компоненты: флажок, соответствующий свойству ProgressiveEncoding, и редактор свойства CompressionQuality (рис. 12.2). Если тип файла снова поменялся и стал равным 1, нужно эти компоненты убрать.

Поможет нам в этом внимательное изучение исходных кодов диалогов, находящихся в модуле EXTDLGS.PAS. Программисты Borland пошли по пути модернизации внешнего вида стандартных диалогов, добавив к ним справа панель для отображения внешнего вида открываемых (записываемых) кар-

тинок. Можно пойти дальше и добавить таким же образом и свои элементы управления.

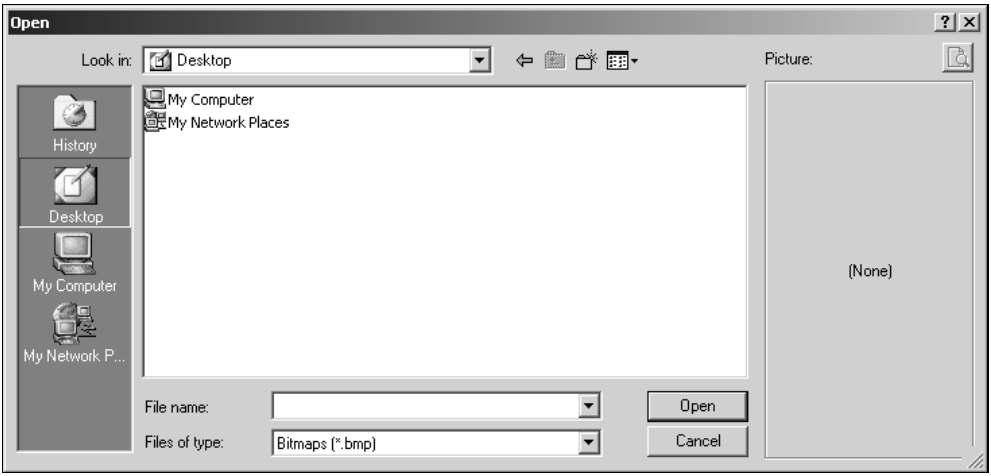


Рис. 12.2. Внешний вид модифицированного компонента `TSavePictureDialog`

Приведенный далее пример ModifDlg — усовершенствованная программа просмотра и сохранения файлов растровой графики, к которым относятся файлы форматов JPEG и BMP (листинг 12.3). Чтобы исключить метафайлы и значки (*.wmf, *.emf, *.ico), соответствующим образом настраиваются фильтры в диалогах открытия и сохранения.

Для изменения размеров диалогового окна нужно отыскать среди входящих в его состав компонентов панель `PicturePanel` (так назвали ее разработчики Borland) и увеличить ее высоту. Следует также поменять и размеры родительских окон. Поскольку они не являются компонентами Delphi (стандартные диалоги являются составными частями Windows), для этой цели применяются функции API `GetWindowRect` и `SetWindowPos`.

Обратите также внимание, что при загрузке используется событие `OnProgress` класса `TGraphic`. В его обработчике информация об объеме проделанной работы отображается на компоненте `ProgressBar1`. Для маленьких картинок обработчик вызывается только в начале и конце операции, пользователь ничего не заметит. Зато при загрузке большого изображения он будет спокоен, видя, что процесс загрузки идет и машина не зависла.

Листинг 12.3. Исходный текст главного модуля программы ModifDlg

```
unit mainUnit;
interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtDlgs, StdCtrls, ComCtrls, ExtCtrls, Buttons;
type
  TForm1 = class(TForm)
    SavePictureDialog1: TSavePictureDialog;
    OpenPictureDialog1: TOpenPictureDialog;
    ScrollBox1: TScrollBox;
    Image1: TImage;
    ProgressBar1: TProgressBar;
    OpenBitBtn: TBitBtn;
    SaveBitBtn: TBitBtn;
    procedure SavePictureDialog1TypeChange(Sender: TObject);
    procedure Image1Progress(Sender: TObject; Stage: TProgressStage;
      PercentDone: Byte; RedrawNow: Boolean; const R: TRect;
      const Msg: String);
    procedure SavePictureDialog1Close(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SavePictureDialog1Show(Sender: TObject);
    procedure OpenBitBtnClick(Sender: TObject);
    procedure SaveBitBtnClick(Sender: TObject);
  private
  public
end;
var
  Form1: TForm1;

implementation
{$R *.DFM}
uses jpeg;
const DeltaH : Integer = 80;
var Quality : TJpegQualityRange;
    ProgressiveEnc : Boolean;
procedure TForm1.OpenBitBtnClick(Sender: TObject);
begin
if OpenPictureDialog1.Execute
 then Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

```
procedure TForm1.SaveBitBtnClick(Sender: TObject);
var ji : TJpegImage;
begin
if SavePictureDialog1.Execute then
 begin
 ji := TJpegImage.Create;
 ji.CompressionQuality := Quality;
 ji.ProgressiveEncoding := ProgressiveEnc;
 ji.Assign(Image1.Picture.Bitmap);
 ji.SaveToFile(SavePictureDialog1.FileName);
 ji.Free;
 end;
end;
procedure TForm1.SavePictureDialog1TypeChange(Sender: TObject);
var ParentHandle:THandle;wRect:TRect;
PicPanel,PaintPanel:TPanel;JEdit : TEdit;
Expanded : boolean;
begin
With Sender as TSavePictureDialog do
begin
 PicPanel := (FindComponent('PicturePanel') as TPanel);
 if not Assigned(PicPanel) then Exit;
 ParentHandle:=GetParent(Handle);

 PaintPanel:=(FindComponent('PaintPanel') as TPanel);
 PaintPanel.Align := alNone;
 Expanded := FindComponent('JLabel') <> nil;
 if FilterIndex >1 then
   begin
   if not Expanded then
    begin
    GetWindowRect(ParentHandle,WRect);
    SetWindowPos(ParentHandle,0,0,0,WRect.Right-WRect.Left,
    WRect.Bottom-WRect.Top+DeltaH,SWP_NOMOVE+SWP_NOZORDER);
    GetWindowRect(Handle,WRect);
    SetWindowPos(handle,0,0,0,WRect.Right-WRect.Left,
    WRect.Bottom-WRect.Top+DeltaH,SWP_NOMOVE+SWP_NOZORDER);
    Expanded:=True;
    PicPanel.Height := PicPanel.Height+DeltaH;
```

```
if FindComponent('JLabel')=nil then
with TLabel.Create(Sender as TSavePictureDialog) do
 begin
  Parent := PicPanel;
  Name := 'JLabel';
  Caption := 'Quality';
  Left := 5;
  Height := 25;
  Top := PaintPanel.Top+PaintPanel.Height+5;
 end;
if FindComponent('JEdit')=nil then
 begin
 JEdit := TEdit.Create(Sender as TSavePictureDialog);
 with JEdit do
  begin
   Parent := PicPanel;
   Name:='JEdit';
   Text := '75';
   Left:=50;Width := 50;
   Height := 25;
   Top := PaintPanel.Top+PaintPanel.Height+5;
  end;
 end;
if FindComponent('JUpDown')=nil then
with TUpDown.Create(Sender as TSavePictureDialog) do
 begin
  Parent := PicPanel;
  Name:='JUpDown';
  Associate := JEdit;
  Increment := 5;
  Min := 1; Max := 100;
  Position := 75;
 end;
if FindComponent('JCheck')=nil then
with TCheckBox.Create(Sender as TSavePictureDialog) do
 begin
  Name:='JCheck';
  Caption:='Progressive Encoding';
```

```
      Parent:=PicPanel;
      Left:=5;Width := PicPanel.Width – 10;
      Height:=25;
      Top := PaintPanel.Top+PaintPanel.Height+35;
     end;
   end;
   end
 else
   SavePictureDialog1Close(Sender);
end;
end;
procedure TForm1.Image1Progress(Sender: TObject; Stage: TProgressStage;
  PercentDone: Byte; RedrawNow: Boolean; const R: TRect;
  const Msg: String);
begin
case Stage of
 psStarting: begin
             Progressbar1.Position := 0;
             Progressbar1.Max := 100;
             end;
 psEnding:   begin
             Progressbar1.Position := 0;
             end;
 psRunning:  begin
             Progressbar1.Position := PercentDone;
             end;
 end;
end;
procedure TForm1.SavePictureDialog1Close(Sender: TObject);
var PicPanel : TPanel; ParentHandle : THandle; WRect : TRect;
begin
With Sender as TSavePictureDialog do
  begin
  PicPanel := (FindComponent('PicturePanel') as TPanel);
  if not Assigned(PicPanel) then Exit;
  ParentHandle:=GetParent(Handle);
  if ParentHandle=0 then Exit;
  if FindComponent('JLabel')<>nil then
```

```
   begin
    FindComponent('JLabel').Free;
    FindComponent('JEdit').Free;
    ProgressiveEnc := (FindComponent('JCheck') as TCheckBox).Checked;
    FindComponent('JCheck').Free;
    Quality := (FindComponent('JUpDown') as TUpDown).Position;
    FindComponent('JUpDown').Free;
    PicPanel.Height:=PicPanel.Height-DeltaH;
    GetWindowRect(Handle,WRect);
    SetWindowPos(Handle,0,0,0,WRect.Right-WRect.Left,
    WRect.Bottom-WRect.Top-DeltaH,SWP_NOMOVE+SWP_NOZORDER);
    GetWindowRect(ParentHandle,WRect);
    SetWindowPos(ParentHandle,0,0,0,WRect.Right-WRect.Left,
    WRect.Bottom-WRect.Top-DeltaH,SWP_NOMOVE+SWP_NOZORDER);
    FilterIndex := 1;
   end;
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
var s: string;
begin
 s :=GraphicFilter(TBitmap)+'|'+GraphicFilter(TJpegImage);
 OpenPictureDialog1.Filter := s;
 SavePictureDialog1.Filter := s;
end;
procedure TForm1.SavePictureDialog1Show(Sender: TObject);
begin
 with Sender as TSavePictureDialog do
 begin
  if FindComponent('JLabel')<>nil then
   begin
    FilterIndex := 2;
    SavePictureDialog1TypeChange(Sender);
   end;
 end;
end;
end.
```

Приведенный пример может послужить толчком, во-первых, к углубленному изучению формата JPEG, а во-вторых, — к модификации стандартных диалогов. На его базе можно создать диалоги открытия аудиозаписей, документов и других специализированных видов файлов.

Класс *TClipboard*

Класс `TClipboard` предоставляет программисту интерфейс с буфером (папкой) обмена (Clipboard) Windows. При включении в проект модуля CLIPBRD.PAS глобальный объект `Clipboard` создается автоматически и доступен приложению в течение всего времени его работы.

Методы открытия и закрытия буфера обмена:

```
procedure Open;
procedure Close;
```

вызываются во всех остальных методах `TClipboard`, поэтому программист редко нуждается в обращении к ним. В объекте ведется счетчик числа обращений к этим функциям, так что соответствующие функции API Windows вызываются только при первом открытии и последнем закрытии.

Очистка содержимого буфера (для всех форматов) производится вызовом метода:

```
procedure Clear;
```

О доступных форматах можно узнать, пользуясь следующими свойствами и методами:

- `property FormatCount: Integer;`

  Содержит число форматов в буфере на данный момент.

- `property Formats[Index: Integer]: Word;`

  Содержит их полный список.

- Функция `function HasFormat(Format: Word): Boolean`. Проверяет, содержится ли в данный момент формат `Format`.

Волею разработчиков различаются способы обмена графической и текстовой информацией через буфер обмена. Рассмотрим их независимо.

Через вызов метода

```
procedure Assign(Source: TPersistent);
```

в буфер обмена помещаются данные классов `TGraphic`, точнее, его потомков — классов `TBitmap` (формат `CF_BITMAP`) и `TMetafile` (`CF_ENHMETAFILE`), а также данные класса `TPicture`. Данные класса `TIcon` не имеют своего формата и с классом `TClipboard` несовместимы.

Допустимо и обратное: в `TClipboard` есть специальные (скрытые) методы для присваивания содержимого объектам классов `TPicture`, `TBitmap` и `TMetafile`. Допустимы выражения вида:

```
MyImage.Picture.Assign(Clipboard);
Clipboard.Assign(MyImage.Picture);
```

Для работы с текстом предназначены методы:

- `function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;`

  Читает текст из буфера обмена в буфер `Buffer`, длина которого ограничена значением `BufSize`. Функция возвращает истинную длину прочитанного текста

- `procedure SetTextBuf(Buffer: PChar);`

  Помещает текст из `Buffer` в буфер обмена в формате CF_TEXT.

Впрочем, можно поступить проще. Свойство

```
property AsText: string;
```

соответствует содержимому буфера обмена в текстовом формате CF_TEXT (приведенному к типу `string`). При отсутствии в буфере данных этого формата возвращается пустая строка.

Методы:

```
function GetAsHandle(Format: Word): THandle;
procedure SetAsHandle(Format: Word; Value: THandle);
```

соответственно читают и пишут данные в буфер в заданном формате `Format`. При чтении возвращается дескриптор находящихся в буфере данных (или 0 при отсутствии данных). Для дальнейшего использования эти данные должны быть скопированы. При записи данные, передаваемые в параметре `Value`, в дальнейшем должны быть уничтожены системой (а не программой пользователя).

Два метода предназначены для обмена компонентами через буфер обмена (в специально зарегистрированном формате `CF_COMPONENT`):

```
function GetComponent(Owner, Parent: TComponent): TComponent;
procedure SetComponent(Component: TComponent);
```

Они используются составными частями среды Delphi.

Класс *TScreen*

Этот компонент представляет свойства дисплея, на котором выполняется приложение. Поскольку экземпляр данного класса только один (он создает-

ся системой при запуске приложения), то большинство методов и свойств имеют информационный характер и недоступны для записи.

Курсор приложения, общий для всех форм, доступен через свойство:

```
property Cursor: TCursor;
```

Часто приходится включать "песочные часы" на время выполнения длительной операции. Правильнее всего это сделать следующим образом:

```
Screen.Cursor := crHourglass;
try
  {Calculations...}
finally
  Screen.Cursor := crDefault;
end;
```

Имеется список всех курсоров. Получить дескриптор курсора с индексом `Index` можно при помощи свойства:

```
property Cursors[Index: Integer]: HCURSOR;
```

Напомним, что индексы зарегистрированных курсоров лежат в диапазоне от −22 (`crSizeAll`) до 0 (`crDefault`).

Рассмотренный далее фрагмент кода (листинг 12.4) при инициализации формы заносит имена всех зарегистрированных в системе курсоров в список `ListBox1`. Затем, при выборе элемента списка, устанавливается соответствующий ему курсор.

Листинг 12.4. Пример работы с курсорами

```
procedure TForm1.FormCreate(Sender: TObject);
type
TGetStrFunc = function(const Value: string): Integer of object;
var
  CursorNames: TStringList;
  AddValue: TGetStrFunc;
begin
  CursorNames := TStringList.Create;
  AddValue := CursorNames.Add;
  GetCursorValues(TGetStrProc(AddValue));
  ListBox1.Items.Assign(CursorNames);
end;
procedure TForm1.ListBox1Click(Sender: TObject);
```

```
begin
 Screen.Cursor := StringToCursor(ListBox1.Items[ListBox1.ItemIndex]);
end;
```

Список курсоров, функции `GetCursorValues`, `StringToCursor` и некоторые другие содержатся в модуле CONTROLS.PAS.

Имена всех установленных в системе шрифтов помещаются в список, определенный в свойстве:

```
property Fonts: TStrings;
```

Компонент сообщает неизменяемые свойства экрана (в данном видеорежиме). Его размеры в пикселах определены в свойствах:

```
property Height: Integer;
property Width: Integer;
```

В последних версиях ОС Microsoft имеется поддержка отображения на нескольких мониторах одновременно. Для этой цели предусмотрены свойства:

```
property MonitorCount: Integer;
property Monitors[Index: Integer]: TMonitor;
```

Каждый компонент `TMonitor` несет информацию о размерах и положении изображения на нем. Образовавшийся же виртуальный рабочий стол характеризуется свойствами:

- `property DesktopLeft: Integer;`
- `property DesktopTop: Integer;`
- `property DesktopWidth: Integer;`
- `property DesktopHeight: Integer;`

Все координаты отсчитываются от верхнего левого угла первого монитора. Если монитор один, значения этих свойств совпадают с `Left`, `Top`, `Width` и `Height`.

Число точек на дюйм дисплея содержится в свойстве:

```
property PixelsPerInch: Integer;
```

При появлении каждая форма заносит себя в список форм глобального объекта `Screen`. Два (доступных только для чтения) свойства дают информацию об этом списке:

```
property Forms[Index: Integer]: TForm;
property FormCount: Integer;
```

Нужно иметь в виду, что в списке указаны только формы, открытые приложением, а не все окна системы.

Следующие два свойства указывают на активную в данный момент форму и ее активный элемент управления:

- `property ActiveControl: TWinControl;`
- `property ActiveForm: TForm;`

При их изменении генерируются, соответственно, события:

- `property OnActiveControlChange: TNotifyEvent;`
- `property OnActiveFormChange: TNotifyEvent;`

Хотя и "некстати", расскажем здесь о свойстве

```
property DefaultKbLayout: HKL;
```

Оно указывает на раскладку клавиатуры, принятую в системе по умолчанию. Часто раскладку клавиатуры нужно переключать программно, чтобы облегчить жизнь пользователю. Так, в приложении, в котором нужно быстро вводить в базу данных большой объем информации на русском и английском языках, такое переключение при смене полей просто необходимо.

Сначала нужно прочитать список имеющихся в системе раскладок и установить нужную:

```
var  RusLayout, EngLayout : THandle;
procedure TMainForm.FormCreate(Sender: TObject);
var Layouts : array[0..7] of THandle;
i,n : Integer;
begin
// считывание раскладок
 RusLayout := 0; EngLayout := 0;
 n := GetKeyboardLayoutList(High(Layouts)+1, Layouts);
 if n>0 then
 for i:=0 to n-1 do
  if LoWord(Layouts[i]) and $FF = LANG_RUSSIAN then RusLayout
:= Layouts[i]
  else if LoWord(Layouts[i]) and $FF  = LANG_ENGLISH then EngLayout
:= Layouts[i];
 // если есть, включим русскую
 if RusLayout<>0 then  ActivateKeyboardLayout(RusLayout,0);
end;
```

Затем при входе в определенное поле (компонент редактирования данных) и выходе из него можно программно сменить раскладку:

```
procedure TMainForm.EditDocSerEnter(Sender: TObject);
begin
 if EngLayout<>0 then ActivateKeyboardLayout(EngLayout,0);
end;
```

```
procedure TMainForm.EditDocSerExit(Sender: TObject);
begin
 if RusLayout<>0 then  ActivateKeyboardLayout(RusLayout,0);
end;
```

Вывод графики с использованием отображаемых файлов

В ряде случаев возможностей, предоставляемых стандартным компонентом, недостаточно. Один из таких случаев — работа с большими и очень большими изображениями (размером до сотен мегабайт). С ними приходится иметь дело в полиграфии, медицине, обработке изображений дистанционного зондирования Земли из космоса и т. п. Здесь `TBitmap` не подходит, так как запрашивает для хранения и преобразования картинки слишком много ресурсов.

Что делать? На помощь следует призвать Windows API, поддерживающий файлы, отображаемые в память (Memory Mapped Files). У них много полезных свойств, но здесь важно только одно из них. При создании битовой карты Windows распределяет для нее часть виртуального адресного пространства. А оно не безгранично — для выделения пятидесяти-ста мегабайт может не хватить размеров файла подкачки, не говоря уже об ОЗУ. Но можно напрямую отобразить файл в виртуальную память, сделав его частью виртуального адресного пространства. В этом случае нашему файлу с изображением будет просто выделен диапазон адресов, которые можно использовать для последующей работы.

Листинг 12.5 иллюстрирует процедуру отображения файла в память и присвоения адреса его данным.

Листинг 12.5. Пример отображения файла в память

```
Var Memory: pByteArray;
ec : Integer;
procedure TForm1.Open1Click(Sender: TObject);
var
 i: integer;
bmFile : pBitmapFileHeader;
bmInfo : pBitmapInfoHeader;
begin
if not OpenDialog1.execute then Exit;
```

```
hf := CreateFile(pChar(OpenDialog1.FileName), GENERIC_READ or
GENERIC_WRITE,
  FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
if hf=INVALID_HANDLE_VALUE then
  begin
   ec:=GetLastError;
   ShowMessage(' File opening error '+IntTostr(ec));
   Exit;
  end;
hm := CreateFileMapping(hf, nil, PAGE_READONLY, 0,0,nil);
if hm=0 then
  begin
   ShowMessage(' File Mapping error %d',[GetLastError]);
   Exit;
  end;
pb := MapViewOfFile(hm, FILE_MAP_READ, 0,0,0);
if pb=nil then
  begin
   ec:=GetLastError;
   ShowMessage('Mapping error '+IntTostr(ec));
   Exit;
  end;
bmFile := pBitmapFileHeader(pb);
if (bmFile^.bfType<>$4D42) then BEGIN Exit; END;
Memory:=@(pb^[bmFile^.bfOffBits]);
bmInfo := @(pb^[SizeOf(TBitmapFileHeader)]);
StrLen:=(((bmInfo^.biWidth*bmInfo^.biBitCount)+31) div 32)*4;
 PaintMe(Self);
end;
```

В этом коде последовательно получены дескрипторы файла (`hf`, с помощью функции `CreateFile`), его отображения в память (`hm`, с помощью функции `CreateFileMapping`) и указатель на отображенные данные (`pb`, посредством `MapViewOfFile`). Не будем вдаваться в детали внутренней реализации битовой карты — графический формат BMP известен достаточно хорошо. Отметим только, что результатом проделанных операций являются структура `bmInfo` типа `TBitmapInfo`, полностью характеризующая битовую карту, и указатель `Memory` на данные битовой карты. Теперь загруженные данные нужно суметь нарисовать на канве, в данном случае на канве объекта `PaintBox`. Делается это так, как показано в листинге 12.6.

Листинг 12.6. Вывод загруженных данных на канве объекта `PaintBox`

```
procedure TForm1.PaintMe(Sender: TObject);
 var OldP : hPalette;i : integer;
begin
if Memory=nil then Exit;
OldP := SelectPalette(PaintBox.Canvas.Handle, Palette, False);
RealizePalette(PaintBox.Canvas.Handle);
SetStretchBltMode(PaintBox.Canvas.Handle, STRETCH_DELETESCANS);
case ViewMode of
vmStretch:
with bminfo^ do
i := Stretch-
DIBits(PaintBox.Canvas.Handle,0,0,PaintBox.Height,PaintBox.Width,
 0,0,biWidth,Abs(biHeight),
 Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS,
 PaintBox.Canvas.CopyMode);
vm1x1:
with bminfo^,PaintBox.ClientRect  do
i := SetDIBitsToDevice(PaintBox.Canvas.Handle,Left,Top,Right-Left,
Bottom-Top,
 Left,Top,Top,Bottom-top,
Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS);
vmZoom:
begin
  with bminfo^,PaintBox.ClientRect  do
 i := StretchDIBits(PaintBox.Canvas.Handle,Left,Top,Right-Left,
Bottom-Top,
 0,0,biWidth,Abs(biHeight),
 Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS,
 PaintBox.Canvas.CopyMode);
end;

end;
if (i=0) or (i=GDI_ERROR) then
   begin
   ec :=GetLastError;
   Form1.Caption := 'Error code '+IntToStr(ec);
   end;
SelectPalette(PaintBox.Canvas.Handle, OldP, False);
end;
```

В зависимости от установленного режима отображения (`vmStretch`, `vmZoom` или `vm1x1`) применяются разные функции WinAPI: `StretchDIBits` или `SetDIBitsToDevice`. Выигрыш в скорости работы приложения особенно ощущается, если загружаемые файлы становятся настолько большими, что должны размещаться в файле подкачки. Наше же приложение не использует его и отображает данные прямо из файла на экран (рис. 12.3).

Рис. 12.3. Этот снимок с метеорологического спутника имеет размер десятки мегабайт

Класс *TAnimate*

В заключение — несколько слов для тех, кто хочет применить в своих программах анимированные (движущиеся) картинки. Самый простой путь для этого — быстрая смена нескольких последовательных битовых карт. Но, во-первых, их еще нужно нарисовать, во-вторых, если у вас достаточно большие картинки или недостаточно мощный компьютер, обязательно будет заметно мерцание, задержки и другие проблемы с выводом на экран. С появ-

лением очередной версии библиотеки элементов управления COMCTL32 гораздо проще применить готовый компонент `TAnimate`.

Этот компонент предназначен для воспроизведения на форме файлов формата AVI (audio-video interleaved; появился впервые с выходом пакета Microsoft Video for Windows).

Альтернативными источниками таких файлов могут послужить:

- файл (с расширением AVI). Его имя нужно задать в свойстве:

```
property FileName: TFileName;
```

- ресурс Windows. Он может быть задан одним из трех свойств:
  - `property ResHandle: THandle;`
  - `property ResID: Integer;`
  - `property ResName: string;`

Наконец, если вы не запаслись своим AVI-файлом, то можете воспользоваться готовым, имеющимся в Windows и иллюстрирующим один из происходящих в системе процессов. Для этого из списка свойства `CommonAVI` нужно выбрать один из вариантов.

Все эти свойства при своей установке обнуляют прочие альтернативные варианты. Запуск ролика начинается при установке свойства `Active` в значение `True`. При этом показываются кадры, начиная с `StartFrame` и до `StopFrame`. Число повторений этой последовательности кадров задается свойством `Repetitions`; если вам нужен бесконечный цикл, установите это свойство в 0.

Что особенно удобно, компонент `TAnimate` снимает проблемы синхронизации показа ролика с другими процессами в системе и вашем приложении. Если свойство `Timers` равно `False`, показ ролика происходит в отдельном программном потоке и никак не влияет на остальное; если оно равно `True`, ролик синхронизируется по внутреннему таймеру. Вы можете привязать к показу ролика, например, проигрывание звука.

Резюме

Графика — не самый маленький и не самый простой раздел программирования в Windows. Описанные в этой главе объекты Delphi сглаживают многие "острые углы", но все равно начинающему без "синяков и шишек" не обойтись. Если у вас есть время и серьезные намерения, посидите над исходным текстом пространства имен `Borland.Vcl.Graphics` — лучшего пособия для самообразования не найти.

ЧАСТЬ III

Использование XML

Глава 13

Основы XML

Формат представления данных XML получил широкое распространение благодаря относительной простоте и универсальности. Документы и данные XML являются самоописываемыми. При помощи XML приложения обмениваются данными, разнообразные протоколы используют XML для представления данных и служебной информации. XML популярен в Интернете и поддерживается всеми современными браузерами.

Программисты в Delphi 2005 при разработке приложений .NET имеют возможность работать с данными в формате XML. В результате разработанные в Delphi приложения могут работать в качестве источника данных не только с традиционными базами данных, электронными таблицами, текстовыми файлами, но и с файлами XML.

В этой главе рассматриваются следующие вопросы.

- Общее представление о XML.
- Основы синтаксиса XML.
- Объектная модель документа DOM.
- Как использовать DOM в приложениях .NET.
- Документ XML.
- Вершина XML.
- Чтение и запись документов XML.

Что такое XML

Аббревиатура XML расшифровывается как eXtensible Markup Language — расширяемый язык разметки. Разрабатывается он консорциумом World Wide Web Consortium (сокращенно W3C, **www.w3.org/XML**). В консорциум входят представители крупнейших фирм-разработчиков, учебных и научных заведений. И хотя решения его публикуются как рекомендательные, мало кто решается на серьезные отступления от этих рекомендаций.

С первого взгляда документ на языке XML похож на своего HTML-собрата. Оба языка ведут историю от общего предка — языка SGML (Standardized Generalized Markup Language). В тексте — те же элементы и их описания, заключенные в угловые скобки (в обоих языках они называются *тегами*). Действительно, с точки зрения синтаксиса языки похожи, разве что XML намного более строг.

Но внешним сходством все и заканчивается. Водораздел проходит в цели применения языков: HTML — это язык *отображения* информации, в то время как XML — это язык ее универсального *описания*.

HTML родился и развивался как язык для Web-браузеров, для "красивого" представления информации, пересылаемой по протоколу HTTP. Сначала появился тег `<FONT>`, позволявший менять начертание и размер шрифта на странице. Потом основные производители браузеров — Netscape и Microsoft — стали лавинообразно добавлять новые теги, соревнуясь в достижении все новых и новых эффектов и весьма мало задумываясь о совместимости. Те из читателей, кто имеет опыт Web-дизайна (а кто не пытался "родить" свою страничку?), знают, как трудно сделать HTML-документ совместимым со всеми типами и версиями браузеров. Даже те, кто этим не занимался, наверняка видели на сайтах реплики вроде "наилучший просмотр в Internet Explorer ver 4.0 и выше" и т. п.

Но сколько бы не было новых конструкций в HTML, суть остается прежней:

- этот язык имел и имеет заранее заданный, фиксированный набор тегов, который разработчик не в состоянии расширить;
- теги HTML не имеют никакого отношения к содержимому информации в документе, они просто показывают, как ее оформить.

В отличие от этого, XML поэтому и называется "расширяемым", что дает возможность любому определить и построить свою систему тегов, описывающую их собственную предметную область. Если вы когда-либо участвовали в работе над большими разнородными информационными системами, то знаете, сколько уходит времени на разработку и согласование форматов обмена данными между приложениями. К тому же, после завершения проекта эти форматы и протоколы зачастую оказываются более никому не нужными — кому охота копаться в горе двоичных структур?

Совсем не так с XML. Язык этот текстовый, причем поддерживает стандарт UNICODE. Это повышает читаемость документов и облегчает их разработку. Кто-то может сказать, что текстовые данные избыточны по сравнению с двоичными и их передача требует большего времени. Это так, но при этом XML может использовать всю инфраструктуру (HTTP, серверы, браузеры), имеющуюся для передачи HTML.

Задача же представления (отображения) XML-документа решается при помощи другого языка — XSL (eXtended Stylesheet Language, расширяемый язык таблиц стилей) и его модификации XSLT (XSL Transformations). Кстати, XSLT описывается на, ну конечно же, на XML.

В заключение вводной части приведем схему работы с XML-документом (рис. 13.1). Помимо самого документа, на входе системы должно присутствовать его описание (определение). Надеемся, из прочитанного ясно, почему в отличие от HTML теги XML-документа нигде не стандартизованы и не описаны, поэтому требуется дополнительная информация о их типах, назначении и связях между ними. Такое определение может представлять собой так называемое DTD (Document Type Definition — определение типа документа), или более новый тип описания — XML-схему. И та, и другая разновидность описания может содержаться как в теле самого документа, так и в отдельном файле, на который документ будет ссылаться.

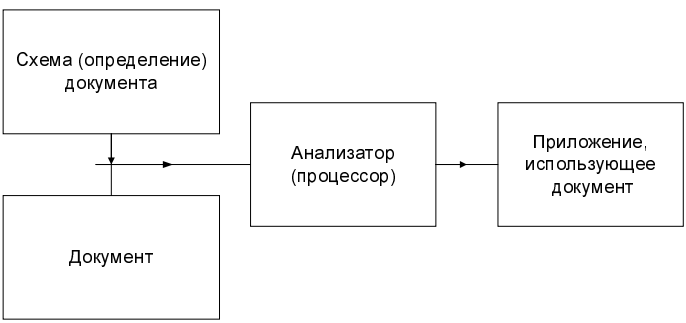


Рис. 13.1. Общая схема работы с XML-документами

Перед началом обработки документа он должен пройти стадию синтаксического анализа и проверки на соответствие DTD/схеме. Такую работу выполняет так называемый *XML-анализатор (parser)*.

Разработчики XML с самого начала задумывались о едином унифицированном способе доступа к документам на новом языке. Тем более, что перед глазами был неудачный пример: Microsoft и Netscape по-разному реализовали объектную модель языка DHTML (Dynamic HTML). Поэтому для XML консорциум W3C разработал и стандартизовал Объектную Модель Документа (Document Object Model, DOM). Она представляет собой набор интерфейсов, соответствующих вершинам (nodes) древовидной структуры XML-документа. После того как DOM-совместимый анализатор загрузил и синтаксически проверил документ, в распоряжении программиста оказываются одни и те же методы и свойства, независимо от операционной системы, языка программирования и реализации анализатора.

Сейчас широко распространены несколько реализаций анализаторов. Для Windows и IE фирмой Microsoft разработан мощный и скоростной анализатор MSXML. Однако, зная о его слишком тесной связи с другими компонентами ОС, и в силу этого — полном отсутствии переносимости, другие поставщики ПО не стали "складывать все яйца в одну корзину". Аналогичные утилиты для других ОС и среды Java имеют Sun, Oracle и IBM.

Наконец, четвертое звено на рисунке — это собственно приложение, предназначенное для работы с XML, принципы их написания и рассматриваются в этой главе. Но прежде рассмотрим первые три звена подробнее.

Основы синтаксиса XML

Базовым понятием XML является элемент. Именно из элементов строится дерево документа. Каждый элемент имеет собственное имя, которое может содержать буквы, цифры, дефисы и символы подчеркивания.

Элемент начинается начальным (открывающим) тегом и завершается закрывающим, между которыми может находиться текстовое значение элемента и другие элементы:

```
<first>Hello, World!</first>
```

Здесь элемент с именем `first` имеет значение "Hello, World!".

Примечание

В языке XML делается различие между строчными и заглавными буквами, в отличие от HTML. Элементы с именами `first`, `First` и `FIRST` — это совершенно разные элементы.

В открывающем теге, помимо имени элемента, могут находиться атрибуты элемента. Атрибуты представляют собой дополнительную информацию, которая может описывать элемент:

```
<salary year="2005" currency="USD">10000</salary>
```

Элемент `salary` (зарплата) имеет значение 10 000, а атрибуты указывают, что выражена эта цифра в долларах и относится к 2005 году.

Элемент может не содержать ни значения, ни атрибутов. В первом случае допустима краткая форма его записи с опусканием закрывающего тега:

```
<dummy/>
```

Это то же самое, что и

```
<dummy></dummy>
```

Элементы могут быть вложенными. Такую структуру обычно применяют, если хотят подчеркнуть, что по смыслу одни элементы являются дочерними для других (листинг 13.1).

Листинг 13.1. Пример XML-документа

```
<?xml version="1.0"?>
<!-- First XML sample document -->
<ProjectTeam project="Super">
       <ProjectManager age="35" certification="MCSD">Mark
       Bose</ProjectManager>
       <Developers>
              <Developer age="25">Jonh Doe</Developer>
              <Developer age="28" certification="MCP">Patrick
              Keegan</Developer>
       </Developers>
       <Testers/>
</ProjectTeam>
```

Элемент `ProjectTeam` имеет три дочерних: `ProjectManager`, `Developers` и `Testers`. Второй из них, перечисляющий разработчиков, содержит в свою очередь два элемента `Developer`. Атрибуты `age` и `certification` несут о них дополнительную информацию.

Более того, элементы должны или не пересекаться, или быть *полностью* вложенными. Частичная вложенность не допускается, конструкции типа

```
<first_element> <second_element></first_element> </second_element>
```

в отличие от HTML, являются ошибочными.

Здесь важно подчеркнуть еще одно важное отличие синтаксиса XML от HTML. В XML *строго* требуется, чтобы открывающему тегу соответствовал закрывающий. Отсутствие последнего является ошибкой, такой документ не будет обработан. В то же время HTML допускает определенную вольность в этом вопросе: там новый тег может отменять действие незакрытого предыдущего. Если вы создаете HTML-документы и при этом хотите задействовать XML-анализаторы, старайтесь таких вольностей все же не допускать.

Перейдем теперь к общей структуре XML-документа, опираясь на содержимое листинга 13.1. XML-документ состоит из трех частей:

- пролога;
- определения (необязательно);
- собственно тела документа, начинающегося с корневого элемента.

Пролог

Пролог может содержать общие для всего документа команды обработки (processing instructions) и комментарии. Команды обработки — это специальные теги, ограничиваемые с двух сторон парами символов `<?` и `?>`. Команды называются так потому, что они представляют собой инструкции анализатору по выбору тех или иных режимов или опций. Как правило, XML-документ начинается с *объявления XML (XML declaration)*:

```
<?xml version="1.0"?>
```

Эта команда обработки, строго говоря, не обязательна, но рекомендована W3C, т. к. указывает на номер версии XML. Кроме обязательного номера версии, объявление XML может содержать описание типа кодировки и признак автономности:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

Признак автономности (`standalone`) указывает, содержит ли документ в себе собственное описание. Если в нем содержатся указания на внешние DTD/схемы, то автономным он не является.

Перед объявлением XML в документе не должно быть никаких символов, не исключая и пробелов — в противном случае анализатор выдаст сообщение об ошибке.

Другая команда обработки, которая часто встречается в прологах документов, — это ссылка на таблицу стилей `xml-stylesheet`, например:

```
<?xml-stylesheet type="text/xsl"
href="http://schemas.biztalk.org/BizTalk/g9boxjl2.xsl"?>
```

Эта команда указывает на таблицу стилей на языке XSL, используемую для отображения документа, скажем, в браузере.

Кроме этих двух, в теле документа могут встречаться и другие команды, в том числе специфичные для конкретного анализатора. Незнакомая команда не должна приводить к исключительной ситуации, она будет просто пропущена анализатором.

Комментарии в тексте XML-документа размечаются символами `<!--` и `-->`. Они могут присутствовать не только в прологе, но и в любом другом месте документа, разумеется, вне других тегов, а не внутри них. Все, что находится между этими парами символов, считается комментарием, анализатор игнорирует его содержимое. Такой комментарий является допустимым:

```
<!-- this is a sample of <element> and <? processing instruction ?> -->
```

Еще один часто встречающийся гость в XML-документах — это так называемые секции `CDATA`. Иногда разработчик хочет поместить в документ содержимое, которое не должно обрабатываться анализатором, скажем, некий

поток данных. Какой смысл отыскивать в этом потоке зарезервированные символы, вроде <, >, и т. п.? Все содержимое, находящееся между <![CDATA[и]]>, анализатор просто пропускает, не проверяя на предмет соответствия синтаксису XML. Но это не означает, что туда можно писать вообще все, что угодно. Формат XML текстовый, и символы с шестнадцатиричными значениями 0x0—0x1F относятся к запрещенным (за исключением символов табуляции (0x9), перевода строки (0xD) и возврата каретки (0xA)). Поэтому если вы хотите поместить в XML двоичные данные, скажем, результат сжатия данных или их шифрования — воспользуйтесь кодировкой Base64 или аналогичной (подробнее об этом — в последующих главах).

Определение

Под определением имеется в виду DTD- или XML-схема. Обе разновидности решают одну и ту же задачу: описать будущую структуру документа — все элементы с их атрибутами, типами данных, связями и уровнями вложенности.

Но почему тогда определение является необязательной составной частью документа?

Для того чтобы документ был обработан анализатором и ваше приложение получило доступ к нему, документ должен удовлетворять определенным требованиям. Согласно спецификации XML, выделяются два типа документов: *правильные (well-formed)* и *состоятельные (valid)*.

Правильный документ не имеет синтаксических ошибок (речь идет об использовании символов <, >, & и др.). Кроме этого, правильный документ удовлетворяет трем условиям.

1. Он содержит по крайней мере один элемент.
2. В нем имеется корневой элемент, уникальный для данного документа.
3. Все остальные элементы должны быть вложены в корневой и/или друг в друга без перекрытия.

Соблюдения условий правильности достаточно для того, чтобы документ был обработан анализатором и вы получили к нему доступ через интерфейсы объектной модели документа (DOM).

Состоятельный документ, во-первых, должен быть правильным, а, во-вторых, он должен полностью соответствовать определению (своему DTD или схеме).

Анализатор от Microsoft позволяет включать или выключать режим проверки на состоятельность. Его свойство `ValidateOnParse` указывает, осуществлять ли такую проверку при загрузке документа. Если вы планируете применить XML для систем автоматизированного документооборота, подчиненных жестким требованиям по формату данных, целесообразно сразу отбраковывать несостоятельные документы.

Изначально структура XML-документа описывалась при помощи определения (DTD), позаимствованного из времен SGML. DTD представляет собой набор тегов, ограниченных символами <! и >. Вот так могло бы выглядеть DTD для нашего документа из листинга 13.1:

```
<!ELEMENT Developer (#PCDATA)>
<!ATTLIST Developer
       age CDATA #REQUIRED
       certification CDATA #IMPLIED
>
<!ELEMENT Developers (Developer+)>
<!ELEMENT ProjectManager (#PCDATA)>
<!ATTLIST ProjectManager
       age CDATA #REQUIRED
       certification CDATA #REQUIRED
>
<!ELEMENT ProjectTeam (ProjectManager, Developers, Testers)>
<!ATTLIST ProjectTeam
       project CDATA #REQUIRED
>
<!ELEMENT Testers EMPTY>
```

Подробное описание DTD выходит за рамки этой книги, тем более, что этот вариант уже сдает свои позиции. Интересующихся деталями отправим к документации W3C.

Идея появления XML-схем проста и красива. Кто-то задался справедливым вопросом: а почему бы не описывать XML-документ на XML? Так и родилось подмножество языка, предназначенное для создания описаний. Это подмножество еще растет и развивается, и пока существуют некоторые его разновидности, такие, как XML Schema definition (XSD) или XML-Data Reduced (XDR).

Тело документа. Корневой элемент

Наконец, корневой элемент (root element), который также носит название элемент документа (document element). Мы уже говорили, что в правильном документе должен обязательно быть один элемент, содержащий в себе все остальные. Это, образно говоря, главный заголовок, делающий документ документом. Его название часто задает предметную область документа.

Увидев в листинге 13.1 название корневого элемента `ProjectTeam`, читатель сразу определит, о чем дальше пойдет речь. А теперь попробуйте добавить

перед или после `ProjectTeam` другой элемент, скажем, `<test/>`. Такой документ сразу будет забракован анализатором.

Предположим, помимо самого документа, в XML-файле содержится также его схема, которая обычно имеет собственный корневой элемент — `<Schema>`. Тогда над двумя существующими элементами (корнем документа и корнем схемы) надстраивают еще один, который часто называют просто `<xml>` (не путать с прологом `<?xml?>`).

Итак, резюмируем рассмотрение структуры XML-документа. Если документ удовлетворяет требованиям правильности и (возможно) состоятельности, то он разбирается анализатором на составные части. После окончания разбора документа программист может получить доступ к любой части, используя парадигму "дерева".

Но нужно обязательно упомянуть здесь о другом способе обработки XML. Параллельно с DOM ведется разработка и другого стандарта, называемого SAX (Simple API for XML). В него заложена совсем другая парадигма. Вместо предоставления разработчику "на блюдечке" уже готового и разбитого на части документа SAX-анализатор генерирует для пользователя систему событий: начало элемента, конец элемента, команда обработки и т. п. Написав и указав обработчики этих событий, программист реализует с помощью них требуемую функциональность.

Такой подход кажется разумным, когда:

- анализируемый документ достаточно велик (мегабайты и более), чтобы останавливать весь процесс в ожидании, пока анализатор полностью обработает его;
- полный анализ документа не требуется; нужные решения можно принять, прочитав только его начальную часть.

В состав анализатора от Microsoft входят объекты, поддерживающие SAX.

Объектная модель документа

Итак, анализатор проверил и загрузил правильный XML-документ. Теперь разработчик может обратиться к нему и получить в свое распоряжение набор интерфейсов, позволяющих работать с документом как с древовидной структурой, — объектную модель документа или DOM. Предоставление DOM — необходимая задача каждого анализатора; к счастью, усилия по стандартизации оказали на XML благотворное воздействие.

Консорциум W3C постоянно работает над усовершенствованием DOM. Основой для работы всех анализаторов стал стандарт DOM level 1, принятый в 1998 году. В настоящий момент действует стандарт на DOM уровня 2 (**http://www.w3.org/TR/DOM-Level-2-Core**). Это не вторая версия одной и

той же модели, а именно второй уровень — с добавлениями и расширениями.

Читатель сразу спросит: а какие именно интерфейсы предоставляет DOM? Какую платформу и язык автор имеет в виду?

Никакую конкретно. С целью достижения переносимости DOM описана на языке описания интерфейсов IDL, разработанном группой OMG (Object Management Group) для спецификации CORBA. IDL позволяет описать только сами интерфейсы как абстрактную (не имеющую реализации) совокупность методов. Консорциум W3C опубликовал поддержку (language bindings) DOM для языков Java и ECMAScript (стандартизованная версия JavaScript от Netscape / JScript от Microsoft).

Мы здесь полагаем, что реализация интерфейсов возлагается на анализатор. Получив от него ссылку на тот или иной интерфейс, разработчик пользуется его методами и свойствами, а всю стоящую за этим работу выполняет анализатор.

Разумеется, на платформе Windows DOM отображается на интерфейсы COM, поэтому в этой книге нам придется работать в знакомых нам терминах COM-объектов. Такое отображение сделано как Microsoft, так и Borland.

Соответственно, программисту на Delphi для работы с DOM предоставлен целый ряд возможностей:

- чистая DOM (интерфейсы `IDOMNode`/`IDOMDocument` и производные от них);
- класс `XMLDocument`;
- обращение к интерфейсам и функциям анализатора напрямую; например, для анализатора MSXML они называются `IXMLDOMNode`/`IXMLDOMDocument`.

Это богатство возможностей не должно вас напугать. Все они базируются на стандартной DOM и немного отличаются синтаксисом и наличием дополнительных функций.

Интерфейсы семейства *IDOMNode*

Основная идея DOM — представить весь результат работы анализатора в виде вершин древовидной структуры, причем каждая вершина будет представлена отдельным объектом. Все типы вершин порождены от общего родительского типа `IDOMNode`. От него порождены интерфейсы для конкретных типов вершин — `IDOMElement`, `IDOMAttr` и т. п.

На самом верхнем уровне, как и полагается, находится вершина, соответствующая всему документу в целом. Ее тип — `IDOMDocument`, который будет рассмотрен позже. Прямым потомком документа является вершина типа `IDOMElement`, соответствующая корневому элементу документа. От корне-

вого элемента исходят вершины, соответствующие его дочерним элементам, и т. д. Атрибуты каждого элемента также представляются в виде отдельного набора его дочерних вершин. Но эти вершины — отдельный список вершин особого типа, которые не могут иметь дочерних.

На рис. 13.2 показано, как будет выглядеть DOM-представление нашего первого XML-документа из листинга 13.1.

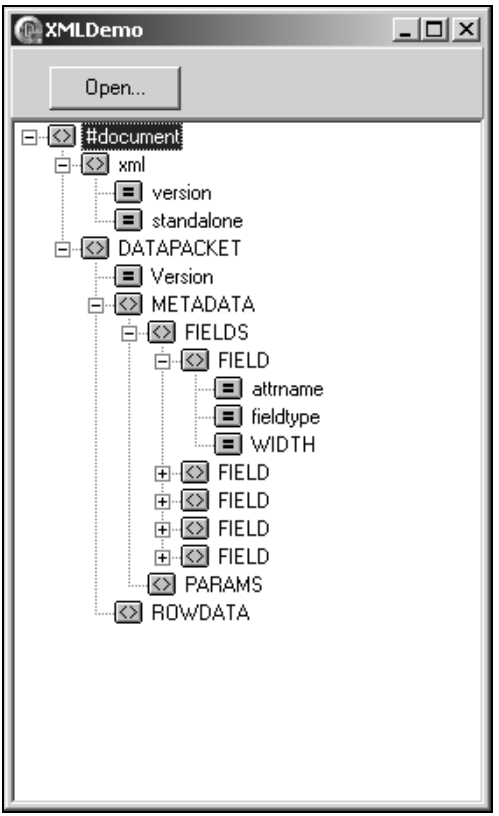


Рис. 13.2. Так выглядит объектная модель нашего первого XML-документа. Рисунок получен при помощи программы, рассматриваемой далее

Здесь видна одна особенность DOM. У каждого элемента, содержащего текст, есть дочерняя вершина, представляющая этот текст. Это типичная причина путаницы, когда пытаются извлечь текст из свойств самого элемента, в то время как его надо искать в дочернем.

Вершине "вообще" соответствует интерфейс `IDOMNode`.

Свойства *nodeType*, *nodeName* и *nodeValue*

Ключ для понимания DOM — разобраться в "трех соснах": свойствах `nodeType`, `nodeName` и `nodeValue`, которые есть у каждой вершины. Самое простое из них `nodeType`, указывающее на тип вершины. А вот в зависимости от его значения интерпретация свойств `nodeName` и `nodeValue` может изменяться. Возможные варианты сведены в табл. 13.1.

Таблица 13.1. *Возможные значения свойств* `nodeType`, `nodeName` *и* `nodeValue` *у интерфейса* `IDOMNode`

Значение `nodeType`		Тип вершины	Значение свойства `NodeName`	Интерпретация значения свойства `NodeValue`	Имя соответствующего интерфейса
Численное	Символьная константа				
1	`ELEMENT_NODE`	Элемент	Имя элемента	`nil`	`IDOMElement`
2	`ATTRIBUTE_NODE`	Атрибут	Имя атрибута	Значение атрибута	`IDOMAttr`
3	`TEXT_NODE`	Текстовая вершина	`#text`	Текст строки	`IDOMText`
4	`CDATA_SECTION_NODE`	Секция `CDATA`	`#cdata-section`	Содержимое секции	`IDOMCDATASection`
5	`ENTITY_REFERENCE_NODE`	Ссылка на сущность	Имя сущности	`nil`	`IDOMEntityReference`
6	`ENTITY_NODE`	Сущность (элемент DTD)	Имя	`nil`	`IDOMEntity`
7	`PROCESSING_INSTRUCTION_NODE`	Команда обработки	Имя команды	Текст команды	`IDOMProcessingInstruction`
8	`COMMENT_NODE`	Комментарий	`#comment`	Содержимое комментария	`IDOMComment`
9	`DOCUMENT_NODE`	Вершина документа	`#document`	`nil`	`IDOMDocument`
10	`DOCUMENT_TYPE_NODE`	Заголовок DTD	Имя DTD	`nil`	`IDOMDocumentType`
11	`DOCUMENT_FRAGMENT_NODE`	Фрагмент	`#document-fragment`	`nil`	`IDOMDocumentFragment`
12	`NOTATION_NODE`	Часть DTD	`#notation`	`nil`	`IDOMNotation`

Подчиненность вершин достаточно очевидна. Вершины TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE и NOTATION_NODE не могут иметь дочерних вершин. Вершина DOCUMENT_TYPE_NODE, соответствующая заголовку DTD, может иметь в подчинении одну или несколько вершин типа NOTATION_NODE или ENTITY_NODE.

Элементы могут содержать другие элементы, атрибуты, комментарии, инструкции обработки, секции CDATA, ссылки и текстовую вершину (одну). Вершина документа может содержать все то же самое, но дочерний элемент может быть только один (корневой).

Мы не будем вдаваться в детальное описание всех двенадцати типов вершин, определенных в DOM. Часть из них относится к DTD, этот стандарт уже почти вытеснен XML-схемами. Другие просто встречаются редко. Те же, с которыми придется работать в последующих главах, будут рассмотрены подробнее.

Свойства и методы, управляющие другими вершинами

Чтобы ориентироваться в структуре дерева, у IDOMNode имеются:

- ссылки на документ и родительскую вершину (свойства ownerDocument и ParentNode);
- ссылки на соседние вершины того же уровня (previousSibling, nextSibling);
- девять методов и свойств для работы с дочерними вершинами;
- кроме того, в отдельную структуру выделены атрибуты (свойство attributes).

В табл. 13.2 — краткое перечисление этих свойств и методов.

Таблица 13.2. *Методы и свойства интерфейса* IDOMNode

Название	Назначение
property ownerDocument: IDOMDocument;	Ссылка на документ, содержащий эту вершину
property parentNode: IDOMNode	Родительская вершина
property previousSibling: IDOMNode;	Предыдущая вершина того же уровня
property nextSibling: IDOMNode;	Следующая вершина того же уровня
function hasChildNodes: WordBool;	Признак того, имеются ли дочерние вершины

Таблица 13.2 *(окончание)*

Название	Назначение
`property childNodes: IDOMNodeList;`	Список дочерних вершин
`property firstChild: IDOMNode;`	Первая дочерняя вершина в списке
`property lastChild: IDOMNode;`	Последняя дочерняя вершина в списке
`function replaceChild(const newChild, oldChild: IDOMNode): IDOMNode;`	Заменяет вершину `oldChild` на `newChild` и возвращает ссылку на старую вершину
`function removeChild(const childNode: IDOMNode): IDOMNode;`	Удаляет вершину `childNode` из списка и возвращает ссылку на нее
`function appendChild(const newChild: IDOMNode): IDOMNode;`	Добавляет новую вершину `newChild` в конец списка дочерних вершин
`function insertBefore(const newChild, refChild: IDOMNode): IDOMNode;`	Добавляет новую вершину `newChild` перед вершиной `refChild`
`function cloneNode(deep: WordBool): IDOMNode;`	Копирует текущую вершину, при этом родительская вершина не устанавливается. Параметр `deep` задает режим клонирования дочерних вершин; при `True` все они также создаются
`property attributes: IDOMNamedNodeMap;`	Интерфейс, предоставляющий список атрибутов вершины
`procedure normalize;`	Нормализует структуру дочерних вершин, удаляя пустые текстовые вершины
`function supports(const feature, version: DOMString): WordBool;`	Поддерживает ли данная вершина в данной реализации DOM заданное свойство `feature` версии `version`

Интерфейсы, хранящие списки дочерних вершин (`IDOMNodeList`) и атрибутов (`IDOMNamedNodeMap`), имеют сходные возможности доступа к своим элементам (`items`):

```
property item[index: Integer]: IDOMNode;
property length: Integer;
```

Впрочем, поскольку перечисленные в табл. 13.2 функции добавления/удаления дочерних вершин с атрибутами не работают, то у `IDOMNamedNodeMap` есть еще и аналогичная функциональность для атрибутов.

Просмотрев имеющиеся функции, вы наверняка заметили, что среди них не нашлось места функциям поиска. Неужели никто не додумался до этого раньше? Ответ: конечно, эта идея приходила в голову разработчикам, и не раз. Но спецификация поиска не входит в стандартный DOM и описывается отдельными документами. Мы обязательно будем об этом говорить далее.

Пространства имен

Имена элементов и атрибутов в XML записываются в текстовом виде. Это большое преимущество — документы являются понятными и удобочитаемыми. Но недостатки, как говорится, — это продолжения достоинств. Предположим, мы объединяем два документа, один из которых описывает список поставщиков продукции, а второй — поставляемые товары. Если один автор документа выбрал для элемента, например, имя `name` или `class`, и второй сделал то же самое, то как отличить имя поставщика от имени товара?

У элементов и их атрибутов имя теперь может являться сочетанием двух составляющих: префикса и собственно локального имени. К примеру, в имени элемента

```
<z:row FullName='GUADELOUPE' Code2='GP' Code3='GLP' NumCode='312'/>
```

`z` является префиксом, а `row` — локальным именем. В нашем случае с товарами и поставщиками теперь можно пользоваться именами `vendor:name` и `product:name`.

Префикс означает принадлежность имени к пространству имен. Пространство имен задается конструкцией:

```
xmlns:[префикс]=" URI пространства имен"
```

Обычно все используемые в документе пространства имен перечисляются в качестве атрибутов корневого элемента. Вот так, например, выглядит корневой элемент набора записей ADO (ADO recordset), сохраненный в виде XML-документа:

```
<xml xmlns:s='uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882'
       xmlns:dt='uuid:C2F41010-65B3-11d1-A29F-00AA00C14882'
       xmlns:rs='urn:schemas-microsoft-com:rowset'
       xmlns:z='#RowsetSchema'>
```

Начиная с модели DOM level 2, в стандарте появилось понятие пространства имен (namespace) и связанные с ним методы и свойства (табл. 13.3).

Таблица 13.3. Свойства и методы пространства имен

Название	Назначение
`property namespaceURI: DOMString;`	URI пространства имен
`property prefix: DOMString;`	Префикс имени вершины
`property localName: DOMString`	Локальное имя вершины

Интерфейс *IDOMDocument*

Этот интерфейс соответствует всему открытому и обработанному анализатором документу. Неудивительно, что в этом интерфейсе собраны методы, относящиеся к верхнему уровню его обработки.

Свойство

```
property documentElement: IDOMElement;
```

указывает на корневой элемент документа; в нашем примере это `ProjectTeam`. Другое свойство

```
property doctype: IDOMDocumentType;
```

дает доступ к DTD документа, если таковое описание в нем содержалось.

Целая группа методов:

```
function createElement(const tagName: DOMString): IDOMElement;
function createDocumentFragment: IDOMDocumentFragment;
function createTextNode(const data: DOMString): IDOMText;
function createComment(const data: DOMString): IDOMComment;
function createCDATASection(const data: DOMString): IDOMCDATASection;
function createProcessingInstruction(const target, data: DOMString):
IDOMProcessingInstruction;
function createAttribute(const name: DOMString): IDOMAttr;
function createEntityReference(const name: DOMString):
IDOMEntityReference;
```

создает вершины нужного типа. При этом они являются обособленными (standalone) и не попадают в дочерний список ни к одной из уже существующих вершин.

Пример создания приложения, использующего модель DOM

Однако теоретическая часть рассказа затянулась, пора попробовать XML "на зуб". XML с его иерархиями вершин напоминает древовидную структуру —

давайте же попытаемся загрузить документ в компонент `TreeView`, чтобы получить наглядное представление.

Поместите на форму компонент `TTreeView`, а также кнопку и диалог открытия файлов, свойство `Filter` у которого настройте на расширение *.xml. Добавьте список картинок `ImageList` с двумя картинками внутри, т. к. нам потребуется различать между собой элементы и атрибуты.

Далее, создайте процедуру `DOMNode2TreeViewNode` и назначьте событию кнопки `OnClick` следующий код (листинг 13.2).

Листинг 13.2. Код приложения XMLDemo1

```
procedure TWinForm1.DOMNode2TreeViewNode(aDOMNode: XMLNode; parentNode:
TreeNode );
var i: Integer;
    ThisNode : TreeNode;
begin
  if not Assigned(ADOMNode)
    then Exit;
  ThisNode := tvXML.Nodes.Add(ADOMNode.Name);
  if not Assigned(ThisNode)
    then Exit;
  ThisNode.ImageIndex := 0;
  if Assigned(ADOMNode.Attributes) then
    for i := 0 to ADOMNode.Attributes.Count - 1  do
      with ThisNode.Nodes.add(ADOMNode.Attributes.Item(i).Name) do
        ImageIndex := 1;
  ThisNode.SelectedImageIndex := ThisNode.ImageIndex;
  if Assigned(ADOMNode.ChildNodes) then
    for i := 0 to ADOMNode.ChildNodes.Count - 1  do
      DOMNode2TreeViewNode(ADOMNode.ChildNodes[i], ThisNode);
end;
procedure TWinForm1.Button1_Click(sender: System.Object; e:
System.EventArgs);
var XMLDocument1: XMLDocument;
begin
  if OpenDialog1.ShowDialog <> System.Windows.Forms.DialogResult.OK
    then Exit;
  XMLDocument1 := XMLDocument.Create;
  try
    XMLDocument1.Load(OpenDialog1.OpenFile);
```

```
    try
      tvXML.Nodes.Clear;
      DOMNode2TreeViewNode( XMLDocument1.DocumentElement, nil );
    except
      on E: Exception do
      begin
        MessageBox.Show( 'Error message:'#13#10 + E.Message);
      end;
    end;
  finally
    XMLDocument1.Free;
  end;
end;
```

В этом листинге основную работу делает рекурсивная процедура `DOMNode2TreeViewNode`:

- добавляет к дереву вершину, соответствующую текущей вершине (элементу) XML-документа;
- добавляет вершины, соответствующие атрибутам этого элемента, если они имеются;
- рекурсивно вызывает саму себя для всех дочерних вершин.

При этом важную роль играет экземпляр класса `XMLDocument`, о котором мы поговорим далее.

Пример делает свою простую работу — отображает файлы формата XML. Но... потренируйтесь с запуском этой программы — не замечаете ли вы на рисунке множество вершин с именем `#text`? Эти вершины искусственно созданы анализатором, мы упоминали об этом ранее при рассмотрении DOM.

Реализация модели DOM в Delphi

У анализатора MSXML много достоинств: высокая скорость (по разным отзывам, он и сейчас на 20—100% быстрее конкурентов), поддержка новейших изменений в стандартах XML. Но как и многие другие решения от Microsoft, он слишком тесно интегрирован с другими частями ОС, в частности, с Internet Explorer.

Примечание

Анализатор MS XML поставляется с Internet Explorer версий от 4.01, а также со всеми новыми операционными системами начиная от Windows 2000 и многими продуктами.

История изменений MS XML сложна и запутанна. Начиная с версии 2 в нем появилась поддержка черновых версий языка XSL Patterns. По мере совершенствования анализатора менялся и язык, и к моменту выхода версии 3 он поменялся полностью — распался на язык преобразований XSLT и поиска — XPath. Менеджеры Microsoft решили поддерживать и старый, и новый диалекты языка, при этом создав для MSXML 3 два режима работы: заменяющий старую версию и совместный (side-by-side). При этом новые COM-объекты, поддерживающие DOM, еще и получили одинаковые со старой версией программные идентификаторы (ProgID). В результате получилось классическое "вавилонское столпотворение", повергшее в хаос множество проектов.

Затем вышел MS XML версии 4, вроде бы предназначенный для исправления ситуации. Из него недрогнувшей рукой убраны все остатки совместимости с XSL. Он больше не называется просто анализатором, теперь это целая parsing suite. Соответственно продукт "потяжелел" до 5 Мбайт. К тому же он устанавливается только с помощью Windows Installer 2.0, runtime которого не входит даже в Windows. Соответственно, прибавьте к размеру инсталляции вашего клиентского приложения еще 2 Мбайт и кучу трудностей. В общем, картина ясна — с точки зрения импортируемости MS XML является весьма спорным вариантом.

Поэтому разработчики Delphi пошли по правильному пути, предусмотрев множественность реализаций DOM. В комплект поставки Delphi входят модули для использования трех анализаторов: MS XML, OpenXML и IBM XML. При этом MS XML является основным анализатором по умолчанию.

Архитектура XML в Delphi такова, что конкретная специфика анализатора скрыта от пользователя еще одним слоем программных интерфейсов. Все производные от XML технологии в Delphi — SOAP, утилита XMLMapper (технология DataSnap) и другие компоненты — сделаны так, чтобы не зависеть от типа применяемого анализатора.

За гибкость в работе с DOM пришлось заплатить, цена гибкости — сложность. "Бабка за дедку, дедка за репку", — вот что вспоминается при рассмотрении иерархии объектов и интерфейсов.

Создайте новое приложение и поместите на главную форму один-единственный компонент `TXMLDocument`. Вы увидите, что к списку `uses` вашего модуля добавились ссылки еще на четыре модуля: `xmldom`, `XMLIntf`, `msxmldom` и `XMLDoc`. Пришло время разобраться с тем, who is who и какую роль играет каждый из этих модулей.

Модуль *xmldom*

Содержит "чистое", не привязанное к реализации, определение COM-интерфейсов в соответствии с XML DOM 2.0 — интерфейс `IDOMNode` и производные от него, а также поддержку некоторых неофициальных, но общепринятых расширений DOM (см. интерфейс `IDOMNodeEx`).

Содержит механизмы поддержки списка реализаций DOM-анализаторов и регистрации подключаемых вами анализаторов.

Модуль *msxml*

Трансляция на язык Delphi библиотеки типов библиотеки msxml.dll версии 2.0. Содержит описания интерфейсов `IXMLDOMNode`, `IXMLDOMDocument` и т. п.

Если вы хотите использовать расширенные возможности более новых анализаторов от Microsoft, создайте аналогичный модуль путем импорта библиотеки типов, имеющейся в msxml3.dll или msxml4.dll.

Модуль *msxmldom*

Не содержит новых интерфейсов (ура!), но в нем описаны объекты (`TMSDOMNode` и производные), которые поддерживают семейство интерфейсов типа `IDOMNode`. Разумеется, это не "настоящие" объекты, а переходники, осуществляющие вызов интерфейсов анализатора от Microsoft. Кроме того, этот модуль берет на себя всю черновую работу по поиску анализатора MSXML подходящей версии.

Точно такую же роль "прокладок" между Delphi и анализатором играют модули `oxmldom` (OpenXML) и `ibmxmldom` (IBM).

Модуль *XMLIntf*

Содержит описания интерфейсов `IXMLNode` и `IXMLDocument` — предлагаемой Borland надстройки над XML DOM, к тому же сглаживающей некоторые острые углы. Эти интерфейсы непосредственно связаны с компонентом `TXMLDocument`

Модуль *XMLDoc*

Ну, и "на закуску", модуль `XMLDoc` и имеющиеся в нем реализации: класс `TXMLNode` и компонент `TXMLDocument`. Связаны эти классы со своими предшественниками просто и изящно. В интерфейсах `IXMLNode` и `IXMLDocument` имеются свойства `DOMNode`/`DOMDocument` соответственно, к которым и происходит обращение со всеми DOM-вызовами. Таким образом, на уровне компонента `TXMLDocument` — первого, с чем столкнутся потенциальные пользова-

тели этого языка в Delphi, — достигается формальная независимость от поставщика анализатора.

Не скрою, эта "луковица" из как минимум трех слоев могла вас и утомить. Но в данном случае нужно отдать должное фирме Borland, которая постаралась соединить максимум удобств с максимумом совместимости. Если какие-то функциональные возможности вам кажутся излишними, никто не мешает вам использовать анализатор напрямую или даже написать свой.

Интерфейс *IXMLNode* и его отличия от стандарта DOM

Вводя параллельно с интерфейсами DOM свою систему интерфейсов, программисты Borland наверняка задавались тем же вопросом, что и пользователи Delphi: а, собственно, зачем? Очевидно, они пытались сделать применение XML более удобным. По сравнению с DOM в интерфейс вершины `IXMLNode` добавлены следующие полезные нововведения:

- добавились свойства `text` (текст вершины) и `xml` (все содержимое данной вершины и всех ее дочерних вершин на XML);
- унифицирован доступ к списку дочерних вершин (свойство `ChildNodes`) и атрибутов (`AttributeNodes`). Теперь оба списка имеют единый тип `IXMLNodeList`;
- добавлены методы доступа (`accessors`), знакомые Delphi-программистам по объекту `TStrings`.
- Свойство

  ```
  property ChildValues[const IndexOrName: OleVariant]: OleVariant;
  ```

  является свойством вершины по умолчанию. Это псевдомассив, доступ к значениям которого возможен как по индексу (номеру) дочерней вершины, так и по ее имени. Для рассмотренного нами примера XML-документа допустимы все три конструкции:

  - `XMLDocument1.DocumentElement.ChildValues['ProjectManager'];`
  - `XMLDocument1.DocumentElement['ProjectManager'];`
  - `XMLDocument1.DocumentElement[0];`
- Свойство

  ```
  property Attributes[const AttrName: DOMString]: OleVariant;
  ```

  дает доступ к атрибутам элемента по их именам.

О свойстве `Text` следует поговорить отдельно, чтобы избежать потенциально возможных заблуждений.

Значение этого свойства определено не для всех вершин, имеется исключение, и оно касается некоторых разновидностей элементов. Проверить, может ли элемент иметь значащее свойство `Text`, можно при помощи другого свойства

```
property IsTextElement: Boolean;
```

доступного только по чтению.

Итак, правило: свойство `Text` определено для элемента только в том случае, если он имеет ровно одну дочернюю вершину и эта вершина имеет тип `ntText` (TEXT_NODE для вершин DOM). Если у элемента еще нет дочерних вершин, его свойство `Text` доступно, и ему может быть присвоено значение. В этом случае необходимая текстовая вершина создается.

Например, для такого элемента

```
<element>This is the text</element>
```

значение `Text` будет равно `"This is the text"`. А если модернизировать пример следующим образом:

```
<element>This is the text<child>and some additions</child></element>
```

то попытка обращения к `Text` вызовет исключительную ситуацию `EXMLDocError`.

Но как же все-таки добиться нужного результата? В листинге 13.3 приведена часть примера, который будет подробно рассмотрен далее в этом разделе. Для вершины `AXmlNode` типа `IXMLNode` мы устанавливаем текст, невзирая на расставленные перед нами "барьеры". Для этого нужно раздельно обработать три частных случая:

- вершина является текстовой (`IsTextElement=True`), в этом самом простом случае мы просто обновляем свойство `Text` (в данном примере назначаемый текст хранится в `EdValue.Text`;
- у элемента есть дочерние вершины, и первая из них является текстовой. Тогда мы обновляем значение ее свойства `nodeValue`;
- у элемента есть дочерние вершины, но первая из них — не текстовая. В этом случае мы вставляем такую вершину при помощи метода `Insert` и снабжаем ее нужным текстом.

Теоретически возможен и четвертый случай — текстовая вершина есть, но она стоит в списке дочерних вершин не первой. Но анализаторы всегда ставят текст на первое место, а такой вариант возможен только в результате программистских манипуляций. Поэтому мы его не учитываем.

Листинг 13.3. Пример работы с `Text`

```
if AXMLNode.IsTextElement
  then AXMLNode.Text := EdValue.Text
  else with AXMLNode do
  begin
   //text added at first
   if (not HasChildNodes) and (EdValue.Text<>'') then
     ChildNodes.Add( OwnerDocument.CreateNode( EdValue.Text, ntText ) )
   else
   begin
    //already has text
    if ChildNodes[0].NodeType = ntText then
      ChildNodes[0].NodeValue := EdValue.Text
    //has children but not text
    else if EdValue.Text<>'' then
    ChildNodes.Insert(0, OwnerDocument.CreateNode(EdValue.Text,
    ntText ));
   end;
  end;
```

Для остальных типов вершин свойство `Text` не порождает подобных проблем. Его значение просто совпадает с значением свойства `NodeValue`.

Взаимосвязь между всеми интерфейсами

Предложив вам на выбор несколько уровней интерфейсов, нужно рассказать о взаимосвязи между ними. Зачем нужен этот путь от общего к особенному? Унифицированный доступ от Borland хорош, но иногда нужно воспользоваться конкретными возможностями анализатора, не попавшими под унификацию.

Особенно это касается анализатора от Microsoft. При всех недостатках с размером и переносимостью функционально он самый полный. Разработчики из Редмонда постоянно опережают стандарты W3C и добавляют к своей реализации интерфейсов DOM новые, подчас нестандартные возможности. Оставим за скобками спор о том, хорошо это или плохо, важно, что доступ к ним однозначно необходим.

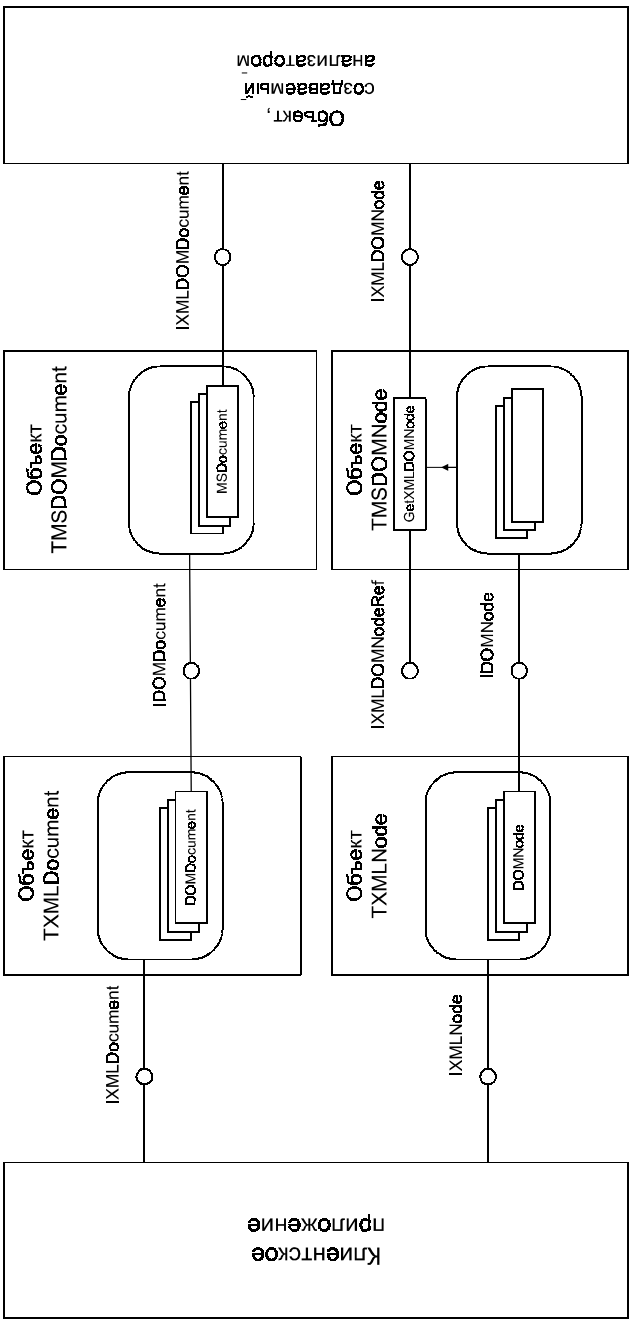


Рис. 13.3. Взаимосвязи интерфейсов и объектов, поддерживающих XML, в Delphi

Итак, как же осуществлена взаимосвязь интерфейсов?

- Переход от `IXMLNode` к `IDOMNode`

  Этот переход прост — каждая вершина первого типа имеет свойство `DOMNode` — явную ссылку на DOM-вершину. Точно так же и в интерфейсе `IXMLDocument` предусмотрена ссылка на `IDOMDocument`.

- Переход от `IDOMNode` к интерфейсу, поддерживаемому анализатором.

  Здесь все менее очевидно. За `IDOMNode` стоит объект Delphi, поддерживающий (implementing) этот интерфейс, так же как и несколько других: `IDOMNodeEx`, `IDOMNodeSelect`, а также `IXMLDOMNodeRef` (в случае анализатора от Microsoft) или `IOXDOMNodeRef` (для OpenXML). `Ref` в конце названия следует понимать буквально: единственное предназначение последних двух интерфейсов — предоставить ссылку на вершину более низкого уровня, непосредственно поддерживаемую анализатором.

  Класс этого объекта в Delphi — `TMSDOMNode` или `TOXDOMNode` соответственно. Обращаясь к `IDOMNode`, вы на самом деле обращаетесь к экземпляру такого объекта, а следовательно, можете применить правила присвоения типов (typecast):

  ```
  MSDOMNode := (DOMNode as IXMLDOMNodeRef).GetXMLDOMNode;
  ```

  или

  ```
  OXDOMNode := (DOMNode as IOXDOMNodeRef).GetDOMNode;
  ```

Взаимосвязи интерфейсов разных уровней для случая анализатора от Microsoft показаны на рис. 13.3.

Нужно иметь в виду, что обратный "подъем" невозможен. Имея вершину типа `IDOMNode`, отыскать соответствующую ей `IXMLNode` очевидными методами нельзя. Поэтому заранее планируйте свой проект так, чтобы работать с интерфейсами только одного уровня.

Загрузка XML

Разработчики ставили перед собой задачу сделать компонент `TXMLDocument` настолько универсальным, насколько возможно. Это в полной мере относится к способам загрузки XML. Вот они:

- из файла. Загрузить из файла можно двумя способами:
  - указав значение свойства `FileName` и затем присвоив свойству `Active` значение `True`. Если значение `FileName` задано еще на этапе разработки, то на этапе выполнения `Active` устанавливается в `True` автоматически;
  - вызвав метод `LoadFromFile` (при этом также изменяются `FileName` и `Active`);

- из потока — вызвав `LoadFromStream`;
- из строки, содержащей текст на XML, — `LoadFromXML`;
- путем присвоения значения свойству `XML`. Для активизации документа в этом случае нужно явно установить `Active = True`.

Заметно, что свойство `Active` играет у `TXMLDocument` ключевую роль. Это, буквально, "выключатель": при активизации документа происходит фактический вызов анализатора. Если он признал документ правильным (и, если требуется, состоятельным), то в этот момент создается иерархия DOM. Инструменты для работы с ней — свойства `DOMDocument`, `documentElement`, `ChildNodes` и другие — получают значения.

И напротив, установка `Active` в `False` уничтожает все внутренние объекты для поддержания DOM и устанавливает указанные свойства в `nil`.

В ряде случаев `Active` устанавливается автоматически, в других случаях вам придется установить его вручную.

Как связывается компонент с XML-документом на этапах разработки и выполнения? Опубликованных свойств у компонента два: `FileName` и `XML`. Взаимосвязь между ними достаточно сложна. Опишем эти правила.

1. На этапе разработки они являются взаимоисключающими — задание одного очищает другое и наоборот.
2. Если задано свойство `FileName`, после запуска приложения значение свойства `XML` будет соответствовать содержимому файла. Но стоит внести изменение в `XML`, как значение свойства `FileName` очищается — это логично, т. к. их содержимое перестает быть синхронным.
3. Изменение свойства XML всегда устанавливает `Active` в `False`. Таким образом компонент застрахован от добавления синтаксически неверных конструкций: при повторной активизации произойдет вызов анализатора, который поможет выявить ошибки.
4. Изменение свойства `FileName` деактивизирует компонент при выполнении двух условий — он был загружен из этого файла и в настоящий момент активен. Если это не так, то документ остается активным, но значение свойства XML очищается.
5. Если во время выполнения изменилось содержимое файла, на который указывает `FileName`, то изменения не отображаются автоматически. Чтобы отследить их, нужно вызвать метод `Refresh`.
6. При загрузке при помощи `LoadFromXML` и `LoadFromStream` значение свойства `XML` соответствует тому, что было загружено; при этом значение `FileName` не изменяется.
7. Если вы хотите автоматически сохранять сделанные вами изменения, установите опцию `doAutoSave` свойства `Options` в `True`. Имеются в виду

изменения, сделанные путем вызовов методов и свойств DOM. Мы уже писали, что DOM-структуры уничтожаются при деактивизации документа. Если упомянутая опция установлена, то текущее состояние всего дерева DOM сохраняется в файле (если оно загружалось оттуда) или в свойстве `XML` (если содержимое было взято из этого свойства). Если же текст XML был взят из строки или потока (пункт 6), компонент справедливо полагает, что сохраняться некуда, и сделанные вами изменения пропадают. Еще раз, речь идет об изменениях, сделанных в DOM (добавление/удаление элементов или их атрибутов, перемещение и т. п.). Если вы просто меняете значение свойства `XML`, см. пункт 3.

8. Наконец, если вы не присваивали значений ни свойству `FileName`, ни свойству `XML`, установка `Active` в `True` создаст для вас пустой XML-документ.

Асинхронная загрузка

Несколько слов об асинхронной загрузке. Такая возможность может быть полезной при работе с XML-документами большого и очень большого размера. У анализатора от Microsoft есть асинхронный режим работы, и поддержка его включена в `TXMLDocument`.

1. В свойстве `ParseOptions` установите опцию `poAsyncLoad`. Тогда во время загрузки четыре раза будет инициализироваться событие `OnAsyncLoad` с параметром `AsyncLoadState`, соответственно равным `loading` (численное значение 1), `loaded` (2), `interactive` (3) или `completed` (4). Это же значение присваивается свойству `AsyncLoadState`. В синхронном режиме событие не возникает, а значение `AsyncLoadState` всегда равно 0 (`Uninitialized`).
2. Разумеется, предлагаемая вам при асинхронной загрузке информация достаточно скупа. Она годится разве что для того, чтобы оповещать пользователя о ходе выполнения и показывать ему индикатор прогресса (progress bar). Если же вы хотите осуществить в процессе загрузки более серьезную обработку, вам следует обратить внимание на SAX-анализаторы.

Функции, создающие экземпляр документа

Если вы хотите создать экземпляр компонента `IXMLDocument` в режиме выполнения, то помимо перечисленных способов, у вас в распоряжении еще несколько функций (модуль `XMLDoc`):

```
function LoadXMLDocument(const FileName: DOMString): IXMLDocument;
function LoadXMLData(const XMLData: DOMString): IXMLDocument; overload;
function LoadXMLData(const XMLData: string): IXMLDocument; overload;
function NewXMLDocument(Version: DOMString = '1.0'): IXMLDocument;
```

Вы получаете в распоряжение интерфейс `IXMLDocument`, но фактически создается экземпляр объекта `TXMLDocument`. После окончания использования его можно уничтожить, просто присвоив `nil` полученному от функции интерфейсу. Для экземпляров `TXMLDocument`, которые вы помещали на форму или модуль данных, присваивать `nil` интерфейсу необязательно. Как и любой другой компонент в этом случае, документ будет иметь владельца (свойство `Owner`) и будет уничтожен библиотекой времени выполнения (RTL) Delphi вместе с содержащей его формой.

Обработка ошибок анализатора

Обработка ошибок также не была пущена на самотек специалистами, разрабатывавшими DOM. Среди интерфейсов объектной модели есть интерфейс `IDOMParseError`, предоставляющий достаточно подробную информацию о случившейся при анализе документа ошибке.

Пара свойств:

```
property errorCode: Integer;
property reason: DOMString;
```

отвечают на вопрос "что?" Разумеется, численные значения кодов ошибок (свойство `errorCode`) и их тексты у каждого анализатора свои. Коды ошибок анализатора от Microsoft можно найти в MSDN, правда, этот вопрос освещен там довольно скупо.

Свойства:

```
property url: DOMString;
property srcText: DOMString;
property line: Integer;
property linePos: Integer;
property filePos: Integer;
```

отвечают на вопрос "где?" Все вместе они позволяют указать точное местоположение ошибки в тексте XML. Названия свойств говорят сами за себя. Разве что, следует указать, что `url` — это адрес документа, а `srcText` — текст строки документа, в котором встретилась ошибка. Свойства `line`, `linePos` и `filePos` — это ее более точные "координаты" в строке/файле.

Нужно добавить, что возникновение ошибок зависит от состояния флага `poValidateOnParse` в свойстве `ParseOptions`. От этого зависит, будет ли документ проверяться только на правильность или на правильность и состоятельность. Проверка на состоятельность зависит еще от местоположения DTD/схемы. Если она находится внутри документа, то вопросов не возникает. Если же схема задана в виде ссылки на внешний документ, то

проверка осуществляется, только если установлена другая опция — poResolveExternals.

Возникает другой вопрос: а где взять этот интерфейс в случае возникновения ошибки анализа? Если вам нужна информация, то Delphi предлагает вам класс исключительной ситуации (exception) EDOMParseError. В этот класс добавлены все те же свойства, что были перечислены для IDOMParseError. Как и любая другая исключительная ситуация, EDOMParseError несет с собой уже готовое сообщение об ошибке в свойстве Message. Программисты Borland поместили в Message конкатенацию значений свойств Reason, Line и первых 40 символов свойства srcText. Если вам нужны более подробные сведения, анализируйте значения остальных свойств. Пример их использования — рассмотренный нами в листинге 13.2 простой загрузчик XML.

Вам уже известно, что вызов анализатора происходит при активизации документа. Это означает, что секции try..except, "отлавливающие" EDOMParseError, должны окружать:

- явную установку свойства Active в True;
- загрузку XML всеми возможными способами.

Но как получить доступ к самому интерфейсу IDOMParseError? Если вы работаете с TXMLDocument и его свойствами, то никак. Если во время анализа документа происходит ошибка, интерфейс IDOMDocument и все остальные интерфейсы DOM обращаются в nil. К IDOMParseError мы вернемся позднее, когда будем рассматривать работу с анализатором напрямую, без "посредничества" Borland.

С компонентом TXMLDocument связаны еще две исключительные ситуации: EXMLDocError и DOMException.

DOMException возникает, когда та или иная возможность не поддерживается текущим анализатором (свойство DOMVendor). Например, такая исключительная ситуация возникает для анализатора OpenXML при попытке обратиться к методам интерфейсов IDOMNodeSelect (SelectNode, SelectNodes) или IDOMNodeEx — TransformNode. Ни язык поиска XPath, ни преобразования XSL/XSLT этим анализатором не поддерживаются.

Вторая ситуация возникает вместе с ошибками, не относящимися к компетенции анализатора. Например, при попытке загрузки из несуществующего файла, модификации документа, доступного только по чтению или создании вершины недопустимого типа. Числового кода ошибки и дополнительных свойств этот класс исключительной ситуации не имеет. И это логично, если EDOMParseError описывает проблемы авторов XML-документов, DOMException — проблемы авторов анализаторов, то EXMLDocError — ваши собственные проблемы как программиста.

Пример использования интерфейсов *IXMLNode* и *IXMLDocument*

Для демонстрации возможностей, предложенных Borland, погрузимся в еще один пример. Давайте посмотрим, как с помощью рассмотренных интерфейсов создать простой редактор XML. Кстати, такой пример может представлять не только познавательный интерес. Свободно распространяемых редакторов XML не так уж и много; проект XML Notepad, разрабатывавшийся в Microsoft, закрыт.

К тому же, процедуры, управляющие вершинами дерева XML (добавление, удаление, перемещение, переименование и т. п.), несколько запутанны даже для опытного программиста. Ничего лучшего, чем редактор, для знакомства с ними придумать нельзя.

Для нашего редактора выберем ставшую уже стандартной схему расположения органов управления. Слева поместим дерево (компонент `TTreeView`), отображающее структуру документа, справа — блокнот `TPageControl` с двумя страницами. Одна служит для редактирования свойства `Text` и списка атрибутов текущей вершины, другая отображает собственно XML (содержимое одноименного свойства `IXMLNode`).

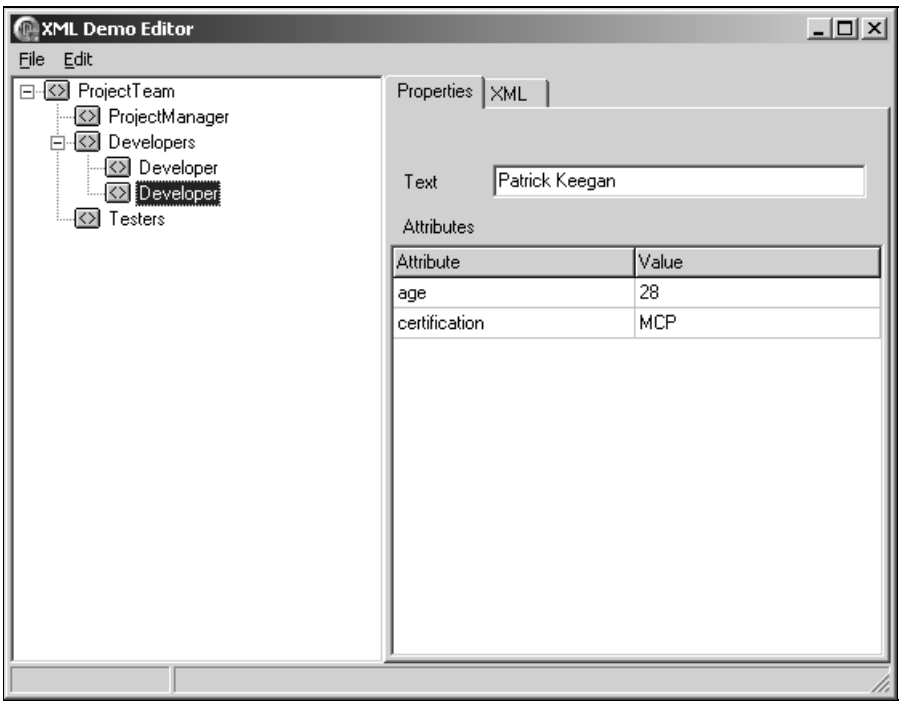


Рис. 13.4. Внешний вид окна демонстрационного редактора XML

Примечание

Хотя свойство XML определено для IXMLNode и формально не зависит от применяемого анализатора, пока оно поддерживается только в том случае если свойство DOMVendor указывает на анализатор от Microsoft.

Внешний вид окна нашего редактора показан на рис. 13.4.

Полный листинг примера (листинг 13.4) не превышает 500 строк — количество, при превышении которого на наш взгляд пример перестает быть примером и становится реальным приложением. Поэтому в нем реализованы только основные функции: чтение/запись, редактирование, работа с буфером обмена. Вы можете добавить функции, необходимые лично вам.

Листинг 13.4. Код XML-редактора

```
unit uMain;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,
  Dialogs, xmldom, XMLIntf, msxmldom, XMLDoc, StdCtrls, ComCtrls,
  ImgList,
  Grids, ValEdit, ExtCtrls, StdActns, ActnList, Menus;
type
  TfmMain = class(TForm)
    Document: TXMLDocument;
    ImageList1: TImageList;
    ActionList1: TActionList;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    N1: TMenuItem;
    SaveAs1: TMenuItem;
    Save1: TMenuItem;
    Open1: TMenuItem;
    New1: TMenuItem;
    Edit1: TMenuItem;
    Paste1: TMenuItem;
    Copy1: TMenuItem;
    Cut1: TMenuItem;
    Delete1: TMenuItem;
```

```
FileOpen1: TFileOpen;
FileSaveAs1: TFileSaveAs;
FileExit1: TFileExit;
EditCut1: TEditCut;
EditCopy1: TEditCopy;
EditPaste1: TEditPaste;
EditDelete1: TEditDelete;
tvXML: TTreeView;
Splitter1: TSplitter;
FileSave1: TAction;
pcEdits: TPageControl;
tsProps: TTabSheet;
edValue: TEdit;
laAttributes: TLabel;
vleAttributes: TValueListEditor;
tsXML: TTabSheet;
meXML: TMemo;
pmAttrEdit: TPopupMenu;
Edit2: TMenuItem;
Paste2: TMenuItem;
Copy2: TMenuItem;
Cut2: TMenuItem;
N7: TMenuItem;
N3: TMenuItem;
DeleteAttr1: TMenuItem;
AddAttr1: TMenuItem;
pmNodes: TPopupMenu;
AddChild: TMenuItem;
DeleteEl: TMenuItem;
AddSibling: TMenuItem;
FileNew1: TAction;
StatusBar1: TStatusBar;
laText: TLabel;
procedure tvXMLChange(Sender: TObject; Node: TTreeNode);
procedure FileSave1Execute(Sender: TObject);
procedure FileOpen1Accept(Sender: TObject);
procedure tvXMLChanging(Sender: TObject; Node: TTreeNode;
  var AllowChange: Boolean);
procedure AddAttr1Click(Sender: TObject);
```

```
    procedure DeleteAttr1Click(Sender: TObject);
    procedure AddChildClick(Sender: TObject);
    procedure tvXMLEdited(Sender: TObject; Node: TTreeNode; var S:
    String);
    procedure AddSiblingClick(Sender: TObject);
    procedure FileNew1Execute(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FileSaveAs1Accept(Sender: TObject);
    procedure EditDelete1Execute(Sender: TObject);
    procedure vleAttributesStringsChange(Sender: TObject);
    procedure edValueChange(Sender: TObject);
    procedure EditCut1Execute(Sender: TObject);
    procedure EditCopy1Execute(Sender: TObject);
    procedure EditPaste1Execute(Sender: TObject);
    procedure ActionList1Update(Action: TBasicAction;
      var Handled: Boolean);
  private
    { Private declarations }
    FDocumentName : string;
    FModified : Boolean;
    procedure SetModified( Value : boolean );
  public
    { Public declarations }
   function GetXMLNode( Node : TTreeNode ): IXMLNode;
   function XMLNodeToTreeViewNode( AXMLNode : IXMLNode; ParentNode :
   TTreeNode ): TTreeNode;
   procedure XMLDocToTreeView( Doc : IXMLDocument; TV : TTreeView );
   property Modified : boolean read FModified write SetModified;
   procedure ClearEdits;
  end;
var
  fmMain: TfmMain;
implementation
{$R *.dfm}
uses ClipBrd;
procedure TfmMain.FormCreate(Sender: TObject);
begin
 FDocumentName := '';
 Modified := False;
end;
```

```
procedure TfmMain.SetModified( Value : boolean );
const ModifiedText : array[False..True] of string = ( '','Modified');
begin
 if value <> FModified then
  begin
   FModified := value;
   Statusbar1.Panels[0].Text := ModifiedText[ FModified ];
  end;
end;
function TfmMain.GetXMLNode( Node : TTreeNode ): IXMLNode;
begin
 Result := nil;
 if not Document.Active
 or not Assigned( Node )
 or not Assigned( Node.Data )
  then Exit;
 try
  Result := IXMLNode( Node.Data );
 except
 end;
end;
procedure TfmMain.FileNew1Execute(Sender: TObject);
begin
 Document.XML.Clear;
 Document.Active := True;
 tvXML.Items.Clear;
 ClearEdits;
 FDocumentName := '';
 Modified := False;
end;
function TfmMain.XMLNodeToTreeViewNode( AXMLNode : IXMLNode; ParentNode :
TTreeNode ): TTreeNode;
var i: Integer;
begin
 Result := nil;
 if not Assigned( AXMLNode ) then Exit;
 try
  Result := tvXML.Items.AddChild( ParentNode, AXMLNode.NodeName );
  Result.ImageIndex := Ord( AXMLNode.NodeType );
```

```
  Result.SelectedIndex := Result.ImageIndex;
  Result.Data := Pointer( AXMLNode );
  for i := 0 to AXMLNode.ChildNodes.Count - 1  do
   if AXMLNode.ChildNodes[i].NodeType <> ntText then //ignore text nodes
     XMLNodeToTreeViewNode( AXMLNode.ChildNodes[i], Result );
 except
 end;
end;
procedure TfmMain.XMLDocToTreeView( Doc : IXMLDocument; tv : TTreeView );
begin
  tv.Items.Clear;
  try
   Screen.Cursor := crHourGlass;
   XMLNodeToTreeViewNode( Doc.DocumentElement, nil );
  finally
   Screen.Cursor := crDefault;
  end;
end;
procedure TfmMain.FileOpen1Accept(Sender: TObject);
begin
 try
  Document.LoadFromFile( FileOpen1.Dialog.FileName );
  XMLDocToTreeView( Document, tvXML );
  FDocumentName := FileOpen1.Dialog.FileName;
  Modified := False;
 except
  on E:EDOMParseError do
   ShowMessage( 'Parser error. Message:'#13#10 + E.Reason );
 end;
end;
procedure TfmMain.FileSave1Execute(Sender: TObject);
begin
 if Document.Active
 and not Document.IsEmptyDoc then
  begin
   Document.SaveToFile( FDocumentName );
  end;
end;
```

```
procedure TfmMain.FileSaveAs1Accept(Sender: TObject);
begin
 FDocumentName := FileSaveAs1.Dialog.FileName;
 FileSave1Execute( Sender );
end;
procedure TfmMain.ClearEdits;
begin
  vleAttributes.Strings.Clear;
  edValue.Text := '';
  meXML.Lines.Clear;
end;
//update TreeNode with XMLNode
procedure TfmMain.tvXMLChange(Sender: TObject; Node: TTreeNode);
var i: Integer;
  AXMLNode : IXMLNode;
begin
  ClearEdits;
  AXMLNode := GetXMLNode( Node );
  if not Assigned( AXMLNode ) then Exit;
  try
    //show text
    if (AXMLNode.NodeType = ntElement)
    and not AXMLNode.IsTextElement then
      begin
      if AXMLNode.HasChildNodes
      and (AXMLNode.ChildNodes[0].NodeType = ntText) then
       edValue.Text := VarToStr( AXMLNode.ChildNodes[0].nodeValue )
      else
       edValue.Text := '';
      end
     else
      edValue.Text := VarToStr( AXMLNode.NodeValue );
    //show attributes
    for i := 0 to AXMLNode.AttributeNodes.Count -1 do
    vleAttributes.strings.Add(
     AXMLNode.AttributeNodes[i].NodeName
      + '=' + VarToStr(AXMLNode.AttributeNodes[i].NodeValue) );
    //show XML
    MeXML.Lines.Text := AXMLNode.XML;
```

```
    Modified := False;
  except
  end;
end;
//update XMLNode with TreeNode
procedure TfmMain.tvXMLChanging(Sender: TObject; Node: TTreeNode;
  var AllowChange: Boolean);
var
  AXMLNode : IXMLNode;
procedure UpdateAttributes( Node : TTreeNode );
var i: Integer;
begin
  AXMLNode := GetXMLNode( Node );
  if not Assigned( AXMLNode ) then Exit;
  try
   AXMLNode.GetAttributeNodes.Clear;
   if vleAttributes.Strings.Count=0 then Exit;
   for i := 1 to vleAttributes.RowCount - 1  do
    if vleAttributes.Keys[i]<>'' then
     AXMLNode.Attributes[ vleAttributes.Keys[i] ] :=
      vleAttributes.Values[ vleAttributes.Keys[i] ];
   except
   end;
end;
procedure UpdateValue( Node : TTreeNode );
var TextNode : IXMLNode;
begin
 AXMLNode := GetXMLNode( Node );
 if not Assigned( AXMLNode ) then Exit;
 if AXMLNode.IsTextElement then
   AXMLNode.Text := EdValue.Text
 else with AXMLNode do
  begin
   //text added at first
   if (not HasChildNodes) and (EdValue.Text<>'') then
     ChildNodes.Add( OwnerDocument.CreateNode( EdValue.Text, ntText ) )
   else
     begin
      //already has text
```

```
      if ChildNodes[0].NodeType = ntText then
        ChildNodes[0].NodeValue := EdValue.Text
      //has children but not text
      else if EdValue.Text<>'' then
        ChildNodes.Insert(0, OwnerDocument.CreateNode( EdValue.Text,
        ntText ) );
     end;
  end;
end;
begin
 if not Modified then  Exit;
 try
  UpdateValue( tvXML.Selected );
  UpdateAttributes( tvXML.Selected );
 except
  AllowChange := False;
 end;
end;
procedure TfmMain.AddAttr1Click(Sender: TObject);
begin
 vleAttributes.InsertRow('','', True );
end;
procedure TfmMain.DeleteAttr1Click(Sender: TObject);
begin
 vleAttributes.DeleteRow( vleAttributes.Row );
end;
procedure TfmMain.AddChildClick(Sender: TObject);
var
 AXMLNode, ChildNode : IXMLNode;
 NodeName : string;
begin
 AXMLNode := GetXMLNode( tvXML.Selected );
 if Not Assigned( AXMLNode ) then Exit;
 if Not InputQuery( Application.Title, 'New node name', NodeName )
  or (NodeName = '') then Exit;
 ChildNode := AXMLNode.AddChild( NodeName );
 XMLNodeToTreeViewNode( ChildNode, tvXML.Selected );
end;
```

```
procedure TfmMain.AddSiblingClick(Sender: TObject);
var
 AXMLNode, SiblingNode : IXMLNode;
 NodeName : string;
begin
 AXMLNode := GetXMLNode( tvXML.Selected );
 if Not Assigned( AXMLNode ) then Exit;
 if Not InputQuery( Application.Title, 'New node name', NodeName )
  or (NodeName = '') then Exit;
 SiblingNode := AXMLNode.ParentNode.AddChild( NodeName );
 XMLNodeToTreeViewNode( SiblingNode, tvXML.Selected.Parent );
end;
 //copied from module XMLDoc
procedure CopyChildNodes(SrcNode, DestNode: IXMLNode);
var
  I: Integer;
  SrcChild, DestChild: IXMLNode;
begin
  for I := 0 to SrcNode.ChildNodes.Count - 1 do
  begin
    SrcChild := SrcNode.ChildNodes[I];
    DestChild := SrcChild.CloneNode(False);
    { Note this fails on documents with DOCTYPE nodes }
    DestNode.ChildNodes.Add(DestChild);
    if SrcChild.HasChildNodes then
      CopyChildNodes(SrcChild, DestChild);
  end;
end;
procedure TfmMain.tvXMLEdited(Sender: TObject; Node: TTreeNode;
  var S: String);
var AXMLNode, NewXMLNode : IXMLNode;
   ParentNode : TTreeNode;
begin
 if s = Node.Text then Exit
 else if s='' then //empty nodes disabled
  begin
   s := Node.Text;
   Exit;
  end;
```

```
 try
  AXMLNode := GetXMLNode( Node );
  if not Assigned( AXMLNode ) then Exit;
  NewXMLNode := AXMLNode.OwnerDocument.CreateElement( s,
AXMLNode.NamespaceURI );
  CopyChildNodes( AXMLNode, NewXMLNode );
  AXMLNode := AXMLNode.ParentNode.ChildNodes.ReplaceNode( AXMLNode,
NewXMLNode );
  AXMLNode := nil;
  ParentNode := Node.Parent;
  Node.Delete;
  tvXML.Selected := XMLNodeToTreeViewNode( NewXMLNode, ParentNode );
 except
 end;
end;
procedure TfmMain.vleAttributesStringsChange(Sender: TObject);
begin
 Modified := True;
end;
procedure TfmMain.edValueChange(Sender: TObject);
begin
 Modified := True;
end;
procedure TfmMain.EditDelete1Execute(Sender: TObject);
var AXMLNode : IXMLNode;
begin
 try
  AXMLNode := GetXMLNode( tvXML.Selected );
  if Not Assigned( AXMLNode ) then Exit;
  AXMLNode.ParentNode.ChildNodes.Remove( AXMLNode );
  tvXML.Selected.Delete;
  if Not Assigned( tvXML.Selected ) then ClearEdits;
 except
 end;
end;
procedure TfmMain.EditCopy1Execute(Sender: TObject);
begin
 try
  Clipboard.AsText := GetXMLNode( tvXML.Selected ).XML;
```

```
 except
 end;
end;
procedure TfmMain.EditCut1Execute(Sender: TObject);
begin
  EditCopy1Execute( Sender );
  EditDelete1Execute( Sender );
end;
procedure TfmMain.EditPaste1Execute(Sender: TObject);
var NewDoc : IXMLDocument;
  AXMLNode, NewNode : IXMLNode;
begin
 if not Clipboard.HasFormat( CF_TEXT ) then Exit;
 try
  try
   NewDoc := LoadXMLData( Clipboard.AsText );
  except
   on E:EDOMParseError do
    ShowMessage( 'Parser error. Message:'#13#10 + E.Reason );
  end;
  if not Assigned( NewDoc ) then Exit;
  //for new( empty ) document
  if tvXML.Items.Count=0 then
   begin
    try
     Document.LoadFromXML( NewDoc.XML.Text );
     XMLDocToTreeView( Document, tvXML );
     FDocumentName := '';
    except
     on E:EDOMParseError do
      ShowMessage( 'Parser error. Message:'#13#10 + E.Reason );
    end;
    Exit;
   end;
  AXMLNode := GetXMLNode( tvXML.Selected );
  if Not Assigned( AXMLNode ) then Exit;
  NewNode := NewDoc.DocumentElement.CloneNode( True );
  AXMLNode.ChildNodes.Add( NewNode );
```

```
  XMLNodeToTreeViewNode( NewNode, tvXML.Selected );
 finally
  NewDoc := nil;
 end;
end;
procedure TfmMain.ActionList1Update(Action: TBasicAction;
  var Handled: Boolean);
begin
 EditCopy1.Enabled := Document.Active and Assigned( tvXML.Selected );
 EditCut1.Enabled := EditCopy1.Enabled;
 EditDelete1.Enabled := EditCopy1.Enabled;
 EditPaste1.Enabled := Clipboard.HasFormat( CF_TEXT );
end;
end.
```

Обратим внимание на несколько особенностей этого примера. Среди методов вершины нет ни одного, позволяющего переименовывать вершины. Это неудивительно, т. к. фундаментальное свойство `NodeName` доступно только по чтению. Для "переименования" нужно:

- создать новую вершину с желаемым именем;
- скопировать в нее все атрибуты и дочерние вершины;
- заменить старую вершину при помощи функции `ReplaceNode`.

Анализатор MSXML, или Microsoft XML Core Services

Рассмотрим анализатор MSXML версии 4. Его основные отличия от предыдущих версий:

- добавлена поддержка языка схем XSD (XML Schema Definition, **www.w3.org/TR/xmlschema-1**);
- добавлена поддержка объектной модели схемы (SOM, Schema Object Model) — нового набора интерфейсов, схожего с DOM, но предназначенного для работы с вершинами схемы;
- убрана поддержка языка XSL (которая поддерживалась в версиях 2 и 3), теперь он полностью вытеснен усовершенствованной версией XSLT;
- удалены независимые от версии анализатора идентификаторы (CLSID и ProgID). Теперь, пытаясь создать экземпляр COM-объекта анализатора, вы должны явно указать, что хотите создать объект нужной версии.

Здесь нужно сделать небольшое отступление и рассказать об истории версий MS XML. Хотя этот анализатор (версии 1.0) впервые появился в Windows 95 OSR2.5 (вместе с Internet Explorer 4.0), специалисты и технические писатели Microsoft любят начинать отсчет с версии 2, впервые поставлявшейся с Office 2000 и Internet Explorer 5 (если быть совсем точным, то 4.01 Service Pack 1). Во всех современных документах Microsoft, касающихся совместимости, MS XML версии 2 — первый из упоминающихся; очевидно, поставляя приложения с использованием MS XML, вы тоже должны озаботиться наличием этой версии. Динамическая библиотека анализатора версии 2, а также последовавшей за ней версии 2.5 (Windows 2000, IE 5.01 и 5.5) называлась msxml.dll.

Промежуточная версия 2.6 поставлялась только с Microsoft SQL Server 2000 и ADO (MDAC) версии 2.6. На этом этапе в библиотеку добавили интерфейс IXMLDOMDocument2, и в Microsoft решили, что самое время переименовать саму динамическую библиотеку в msxml2.dll. Эта версия библиотеки никогда не была доступна для отдельного скачивания и включения в состав дистрибутивов и оказалась самой короткоживущей.

Примечание

Мы упоминаем здесь названия только одной dll, однако это не означает, то анализатор состоит только из нее. В его состав входят еще одна или две библиотеки ресурсов, подлежащих интернационализации. В любом случае, простое копирование и включение в состав дистрибутива нужных dll не гарантирует правильной работы. Microsoft предоставляет распространяемые (redistributable) варианты анализаторов.

Спустя короткое время был выпущен анализатор версии 3 (msxml3.dll). Его основное отличие — поддержка стандарта XSLT. Но мало добавить стандарт — нужно, чтобы с ним работал флагманский XML-продукт от Microsoft — Internet Explorer, который, напомним, поставлялся с MSX ML версии 2. Поэтому в новом анализаторе появилось два режима работы. В режиме совместимости (side-by-side) он регистрировался отдельно, не влияя на прежние версии; таким образом, применение XSLT в IE было недоступно, поскольку, пытаясь создать COM-объект, браузер обращался к анализатору старой версии.

В режиме замены (replace mode) анализатор версии 3 "замыкал" все настройки реестра на себя, полностью вытеснял старый MS XML, и в том числе становился основным анализатором для IE. Переключение из режима в режим производилось при помощи утилиты XmlInst.exe; она поставлялась с анализатором и сейчас доступна для скачивания с сайта Microsoft.

Однако такое вольное обращение с продуктом, выпущенным многомиллионным тиражом, не могло не вызвать шквал проблем. Многие Web-

приложения отказывались работать, пользователи и специалисты путались в режимах работы. К выходу следующей версии было принято решение об отходе. Версия 4 (динамическая библиотека msxml4.dll), как уже упоминалось, больше не поддерживает идентификаторов, независимых от версии.

Примечание

Интересно, что бета-версия MSXML 4 еще поддерживала их. Если вы устанавливали ее на свой компьютер, то обязательно удалите!

Предположим, у вас на компьютере есть MSXML версий 2, 3 и 4 (вполне нормальная ситуация для разработчика), и рассмотрим ситуацию с идентификаторами с точки зрения программиста на Delphi в деталях.

Напомним, что идентификатор класса (CLSID) — это уникальный идентификатор, указывающий на класс COM-объекта, а программный идентификатор (`ProgID`) — его текстовый синоним, придуманный изначально для пользователей Visual Basic.

В Delphi создать объект можно путем вызова функции `CreateCOMObject`:

```
Var Doc : IXMLDOMDocument2;
...
 Doc := CreateCOMObject( CLASS_DOMDocument ) as IXMLDOMDocument2;
```

В этом случае вы получите ссылку на нужный вам интерфейс, поддерживаемый созданным экземпляром объекта с заданным CLSID (в данном примере `CLASS_DOMDocument`).

Можно воспользоваться функцией `CreateOLEObject`, использующей `ProgID`:

```
Var Doc : OleVariant;
...
 Doc := CreateOLEObject( 'MSXML2.DOMDocument' );
```

Она возвращает интерфейс `IDispatch`, поддерживаемый объектом.

Итак, версия 2 анализатора от Microsoft поддерживала объект DOM-документа с идентификатором класса `CLASS_DOMDocument` и программными идентификаторами 'MSXML.DOMDocument', 'Microsoft.XMLDOM' и 'Microsoft.XMLDOM.1.0' (из которых в основном использовался первый).

С появлением в версиях 2.6 и 3 интерфейса IXMLDOMDocument2 появились CLASS_DOMDocument26, CLASS_DOMDocument30 и программные идентификаторы: 'MSXML2.DOMDocument' (независимый от версии), 'MSXML2.DOMDocument.2.6' и 'MSXML2.DOMDocument.3.0'.

Примечание

Цифра 2 в названии 'MSXML2.DOMDocument' означает версию интерфейса, а не версию анализатора. Программные идентификаторы, начинающиеся с 'MSXML2', можно встретить и в версии 3, и в версии 4, но не в версии 2.0.

В режиме замены анализатор версии 3 подменял настройки реестра таким образом, что все независимые от версии идентификаторы — CLASS_DOMDocument, 'MSXML.DOMDocument' и 'MSXML2.DOMDocument' — указывали на msxml3.dll как COM-сервер, создающий требуемые объекты. В режиме совместимости "независимые" идентификаторы (кроме 'MSXML2.DOMDocument') — обращались к старой dll, а те, которые содержали в названии номер версии, — к новой.

Наконец анализатор версии 4 добавляет в реестр CLASS_DOMDocument40 и 'Msxml2.DOMDocument.4.0'. При этом он не трогает настроек прочих версий: например, если вы попытаетесь создать 'MSXML2.DOMDocument', произойдет обращение к msxml3.dll.

Так что если вы совершенно точно знаете, что именно установлено на машине клиента, лучше создавать экземпляр анализатора с нужным номером версии. Если нет, удобно использовать функцию, предлагаемую Borland (модуль msxmldom):

```
function CreateDOMDocument: IXMLDOMDocument;
```

Она последовательно перебирает идентификаторы классов, начиная от 4 до 2, и пытается создать самый новый из возможных объектов.

Анализатор версии 4 еще не поставляется ни с одной операционной системой или другим продуктом от Microsoft. Поэтому, если вы собрались положиться на его возможности, вам придется распространять с вашим продуктом дистрибутив анализатора (около 5 Мбайт) и, скорее всего инсталлятор Windows Installer версии 2 (она также еще не включена в состав ОС от Microsoft).

Как создать экземпляр объекта, поддерживаемого анализатором, и как называется этот объект?

Для получения описаний нужных интерфейсов вы можете подключить уже готовый файл MSXML.PAS, который соответствует версии 2.0. Можно пойти и другим путем и импортировать библиотеку типов анализатора, находящегося на вашей машине (пункт меню Delphi **Project | Import Type library**). Только так вы можете получить доступ к интерфейсам, появившимся в четвертой версии продукта. В диалоге **Import Type Library** ограничьтесь нажатием кнопки **Create Unit**, а не **Install**. В противном случае Delphi превратит в компоненты и установит в категорию ActiveX Палитры инструментов около 40 COM-объектов, 90% из которых вам вряд ли когда-нибудь понадобятся.

В качестве примера рассмотрим взаимодействие двух технологий от Microsoft: XML и ADO. Превратим набор данных ADO в XML-документ, отредактируем его и проведем обратное преобразование.

В ADO, начиная с версии 2.5, появилась возможность загружать и сохранять содержимое курсора (объекта `Recordset`) в виде XML-документа, сопровождаемого своей XDR-схемой.

Если вы не уверены, какие версии ADO устанавливались на вашем компьютере, проще всего открыть редактор реестра regedit.exe и среди дочерних ключей улья `HKEY_CLASSES_ROOT` найти все, начинающиеся на 'ADODB.Recordset'. Если среди них есть 'ADODB.Recordset.2.5' или выше, наверняка нужная вам версия уже установлена.

Итак, убедившись в отсутствии проблем с ADO, мы можем скомпилировать и испытать предлагаемый пример.

Поместив на форме компоненты `TADOConnection` и `TADODataset`, свяжем их с файлом DBDEMOS.MDB, поставляемым с Delphi в качестве примера баз данных от Microsoft Access (он устанавливается по умолчанию в папку ..\Program Files\Common Files\Borland Shared\Data). Еще два компонента — `TDataSource` и `TDBGrid` — понадобятся нам для отображения содержимого этой базы в "традиционном" виде.

Выпадающий список будет содержать список всех таблиц в подключенной базе (это хорошо знакомые всем примеры Counrty, employees, parts и т. п.).

Наконец, компонент `TMemo` будет обеспечивать хранение текста XML, а две кнопки — собственно преобразование (рис. 13.5). Приведем в листинге 13.5 только код преобразования — полный код примера имеется на прилагаемом к книге диске.

Листинг 13.5. Преобразование ADO Recordset в XML и обратно

```
const
 SParserError =
  'Parser error. Message: %s'#13#10'URL: %s; src : %s;'#13#10'Line: %d,
   LinePos: %d';
procedure TForm1.btnToXMLClick(Sender: TObject);
var
 rs : OleVariant;
 Doc : IXMLDOMDocument;
begin
 try
 Doc := CreateCOMObject( CLASS_DOMDocument40 ) as IXMLDOMDocument;
```

```
 try
  rs := ADODataSet1.Recordset;
  rs.Save( Doc, adPersistXML );
  if Doc.parseError.errorCode <> 0 then
   with Doc.parseError do
    begin
     Memo1.Lines.Clear;
     ShowMessage( Format( SParserError,
     [Reason, URL, srcText, Line, LinePos ]
     ) );
    end
  else
   Memo1.Text := Doc.xml ;
 finally
  rs := UnAssigned;
  Doc := nil;
 end;
 except
  on E:Exception do
    ShowMessage( E.ClassName + #13#10 + E.Message );
 end;
end;
procedure TForm1.btnFromXMLClick(Sender: TObject);
var
 rs : _Recordset;
 Doc : IXMLDOMDocument;
begin
 try
  try
  Doc := CreateCOMObject( CLASS_DOMDocument40 ) as IXMLDOMDocument;
  if not Doc.loadXML( Memo1.Text ) then
   with Doc.parseError do
    begin
     Memo1.Lines.Clear;
     ShowMessage( Format(  SParserError,
     [Reason, URL, srcText, Line, LinePos ]
     ) );
     Exit;
    end;
```

```
  rs := CreateCOMObject( CLASS_Recordset ) as _Recordset;
  rs.Open( Doc, EmptyParam,  adOpenUnspecified, adLockUnspecified,
adCmdFile );
  ADODataSet1.Recordset := rs;
 except
  on E:Exception do
    ShowMessage( E.ClassName + #13#10 + E.Message );
 end;
 finally
  rs := nil;
  Doc := nil;
 end;
end;
```

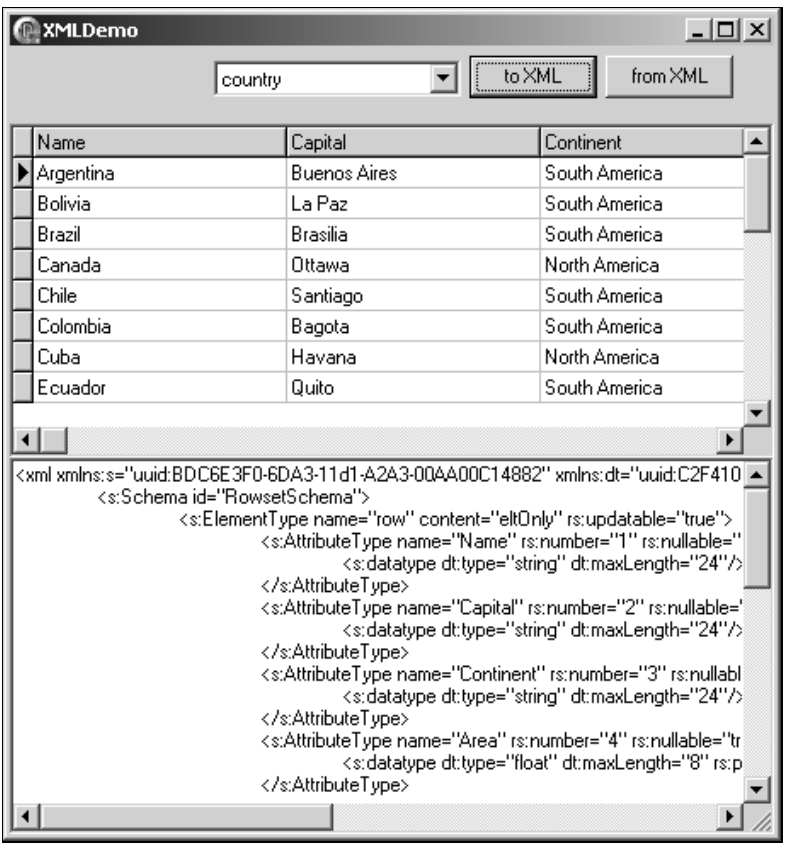


Рис. 13.5. Внешний вид главной формы примера XMLDemo0

Нажав кнопку **to XML**, вы превратите набор данных в документ. Он доступен для редактирования. Добавьте в него еще один элемент с именем `z:row` — каждый такой элемент соответствует записи в выбранной таблице. Отредактируйте атрибуты этого элемента — они соответствуют полям таблицы. Другие значения атрибутов нужны, чтобы избежать потенциально возможных коллизий в таблице при обратном превращении.

Наконец, нажав кнопку **From XML**, вы инициируете обратное превращение. Внесенные вами исправления отобразятся в компоненте `TDBGrid`, но на диск записываться не будут.

Одна из важных для понимания особенностей данного примера — работа с интерфейсом `IStream`. Реализация XML-документа от Microsoft широко используется этим интерфейсом: во-первых, она сама поддерживает его, во-вторых, умеет читать и писать XML в любые объекты, которые поддерживают такой интерфейс.

Набор данных ADO также умеет работать с `IStream`. Методы `Open` и `Save` предусматривают передачу в качестве параметра указателя на объект, поддерживающий такой интерфейс. Единственное неудобство в Delphi заключается в том, что в описании интерфейса `Recordset` такой полиморфизм метода `Save` еще не известен. Выход из положения прост: присвоить указатель на `Recordset` переменной типа `OleVariant` и затем вызвать `Save` посредством dispatch-интерфейса, как это сделано в коде метода `btnToXMLClick`.

Примечание

Еще более радикальный выход из положения — отбросить `TADODataSet` и другие компоненты Delphi и работать с объектами ADO напрямую, импортировав соответствующую TLB. Однако в этом случае вы рискуете вместе с водой выплеснуть и ребенка — унифицированный доступ к данным, на который программисты Borland потратили немало времени (и не зря!). Компромиссный путь — продолжить использование модуля ADODB и имеющихся в нем компонентов, но при этом заменить модуль `ADOInt` на соответствующий модуль, полученный путем импорта TLB от ADO 2.5 и выше. Впрочем, переделка в этом случае затронет более десятка методов и событий и потребует от вас соответствующей квалификации.

Обратите внимание, что в примере нет ни одного модуля по работе XML из состава Delphi (`xmldom`, `msxml`, `msxmldom`, `xmlintf`, `xmldoc`).

Резюме

Формат представления данных XML универсален и широко используется в современных приложениях. Объектная модель, реализованная в .NET и Delphi, почти полностью повторяет объектную модель DOM.

Представление данных в XML базируется на понятии узлов — составных частей XML. За функционал абстрактного узла XML отвечает класс `XmlNode`. На его основе разработаны дочерние классы, представляющие все разнообразие типов узлов в XML. Важнейшими из них являются элементы и атрибуты, реализованные в классах `XmlElement` и `XmlAttribute` соответственно.

Отдельная самостоятельная структура данных XML называется документом XML. Класс `TXmlDocument` инкапсулирует документ XML. Как правило, один файл XML соответствует одному документу. Впрочем, документ XML это только один из типов узлов XML, обладающий уникальными свойствами и методами.

Помимо собственно документов XML, в распоряжении разработчиков имеются и другие возможности XML: описания документов, DTD, схемы документов XSL и язык поиска XPath.

Глава 14

Преобразование данных в формате XML

Программисту доступен набор инструментов и компонентов, которые обеспечивают прямое и обратное преобразование данных XML в формат пакетов данных Delphi и обратно.

В результате разработанные в Delphi приложения могут использовать в качестве источника данных не только традиционные базы данных, электронные таблицы, текстовые файлы, но и файлы XML.

Переход с одного источника данных на другой очень прост. На этапе разработки и подготовки данных есть специальная утилита — XML Mapper. Во время выполнения всю необходимую работу выполнят компоненты преобразования данных категории **Data Access** Палитры инструментов.

В этой главе рассматриваются следующие вопросы.

- Что необходимо для работы с данными XML.
- Каким требованиям должны удовлетворять схемы XML.
- Взаимодействие форматов XML и пакетов данных Delphi.
- Утилита XML Mapper.
- Компонент преобразования данных XML — `TXMLTransform`.

Преобразование данных в формате XML

В Delphi реализован механизм преобразования данных в формате XML во внутренний формат пакетов данных. Пакеты данных широко применяются при передаче данных между клиентом и сервером в распределенных приложениях баз данных. (Речь об этом пойдет далее в этой книге.)

Почему выбран именно такой способ работы с данными XML?

Ответ прост. Благодаря пакетам данных набор компонентов трансформации XML легко интегрируется с практически любыми типами приложений Delphi. Данные XML преобразуются для использования в компонентах клиент-

ских наборов данных, поэтому разработчик автоматически получает в свои руки мощный инструмент просмотра, редактирования и сохранения данных XML. И при этом ему не нужно изучать новые технологии, а достаточно уже хорошо знакомых.

В конце этой главы мы рассмотрим пример, в котором буквально в несколько действий и при помощи немногих компонентов (большинство из которых описывается в других главах этой книги) будет создано приложение, позволяющее не только просматривать и редактировать данные XML, но и опубликовать их на Web-странице.

А сейчас приступим к детальному изучению механизма работы с данными XML.

Схема преобразования данных XML

Итак, попробуем рассмотреть следующую задачу. В качестве исходного выступает файл XML, содержащий некоторые данные. (Подробнее о формате XML *см. главу 13.*) Необходимо преобразовать эти данные в формат пакета данных Delphi.

Во-первых, очевидно, что документ XML должен обладать упорядоченной структурой, которая позволит представить данные в табличном виде.

Во-вторых, необходимо сформировать структуру пакета данных, которая бы смогла адекватно представить преобразованные данные.

И в-третьих, необходимо сопоставить каждый элемент XML с нужным полем пакета данных.

Таким образом, вырисовывается общая схема трансформации данных XML в формат пакета данных (рис. 14.1).

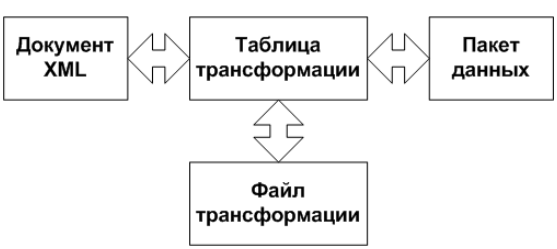


Рис. 14.1. Схема преобразования данных XML в формат пакета данных Delphi

На основе исходного файла XML при помощи средств Delphi создается специальный *трансформационный файл*, в котором описываются связи между узлами XML и полями пакета данных (попробуйте угадать, на каком языке

создается это файл, — конечно с помощью XML), а также структура результирующего документа.

Если преобразование будет выполнено в новый пакет данных, то связывание можно осуществить один к одному, т. е. для каждого узла XML создать поле пакета данных. Если пакет данных уже существует, то необходимо связать узлы XML только с существующими полями пакета.

После преобразования пакет данных можно использовать в приложении или сохранить в файле. Для этого существует метод

```
procedure SaveToFile(const FileName: string = ''; Format
TDataPacketFormat=dfBinary);
```

компонента `TClientDataSet` клиентского набора данных. Формат файла зависит от значения параметра `Format`. При значении `dfXML` данные будут сохранены в формате XML.

Следует отметить, что рассмотренная здесь и реализованная в Delphi схема позволяет осуществлять трансформацию данных как в направлении XML — пакет данных, так и обратно, пакет данных — XML. Но для каждого варианта необходимо создавать собственный трансформационный файл.

Формат пакета данных Delphi

Пакеты данных Delphi необходимы при обмене информацией между клиентским набором данных и компонентом-провайдером. Подробнее об этом *см. главу 28*. Доступ к пакету данных обычно осуществляется через свойство типа `OleVariant`. Пакет данных может быть сохранен в файле XML при помощи метода

```
procedure SaveToFile(const FileName: string = ''; Format
TDataPacketFormat=dfBinary);
```

компонента `TClientDataSet` клиентского набора данных, но для этого в параметре `Format` необходимо задать значение `dfXML`. При значении параметра `dfBinary` пакет сохраняется в бинарном виде в файле с расширением cdr.

Например, листинг 14.1 иллюстрирует, как выглядит пакет данных для таблицы Country демонстрационной базы данных Delphi.

Листинг 14.1. Файл XML пакета данных для таблицы Country

```
<?xml version="1.0" standalone="yes"?>
<DATAPACKET Version="2.0">
 <METADATA>
  <FIELDS>
```

```
    <FIELD attrname="Name" fieldtype="string" WIDTH="24"/>
    <FIELD attrname="Capital" fieldtype="string" WIDTH="12"/>
    <FIELD attrname="Continent" fieldtype="string" WIDTH="13"/>
    <FIELD attrname="Area" fieldtype="string" WIDTH="7"/>
    <FIELD attrname="Population" fieldtype="string" WIDTH="9"/>
   </FIELDS>
   <PARAMS/>
  </METADATA>
  <ROWDATA/>
</DATAPACKET>
```

Структура XML, описывающая пакет данных, обязательно должна содержать элемент

```
<DATAPACKET Version="2.0">
```

и представляет собой описание полей, включенных в состав набора данных. Для этого служит элемент `<FIELDS>`. Если в пакете имеются реальные данные, то они описываются в элементе `<ROWDATA>` в виде записей. Для каждой записи создается отдельный элемент `<ROW>`.

Инструментарий преобразования данных XML

Для создания трансформационных файлов и переноса данных на этапе разработки в состав Delphi включена утилита XML Mapper.

Функции преобразования файлов XML в приложениях доступны через набор компонентов со страницы **Data Access** Палитры инструментов.

Непосредственно для трансформации данных предназначен компонент `TXMLTransform`. Он может осуществлять перевод данных XML в пакет данных, так и обратно.

Компонент-провайдер `TXMLTransformProvider` обеспечивает доступ к файлу с данными XML. Он представляет собой мост между файлом XML и клиентским набором данных, формирует и передает пакеты данных клиентскому приложению, а также принимает от него сделанные изменения.

Компонент `TXMLTransformClient` является контейнером для компонента `TXMLTransform` и обеспечивает преобразование данных, полученных от компонента-провайдера `TXMLTransformProvider`.

Кроме этого, на странице **Internet Express** Палитры инструментов имеется компонент `TXMLBroker`, который получает пакеты данных от компонента-провайдера (в том числе от удаленного сервера приложения) и преобразует их в формат XML для компонентов продюсеров.

А теперь подробнее познакомимся с перечисленными инструментами и посмотрим, как они действуют при создании приложений.

Утилита XML Mapper

Утилита XML Mapper вызывается из меню **Tools | XML Mapper**. Она обеспечивает создание трансформационных файлов, схем XML, файлов с данными (как XML, так и пакетов данных). В дальнейшем эти файлы доступны в приложениях. Например, трансформационные файлы необходимы в работе компонентов, выполняющих преобразование данных.

В соответствии со схемой преобразования данных XML (см. рис. 14.1) окно XML Mapper разделено на три панели (рис. 14.2).

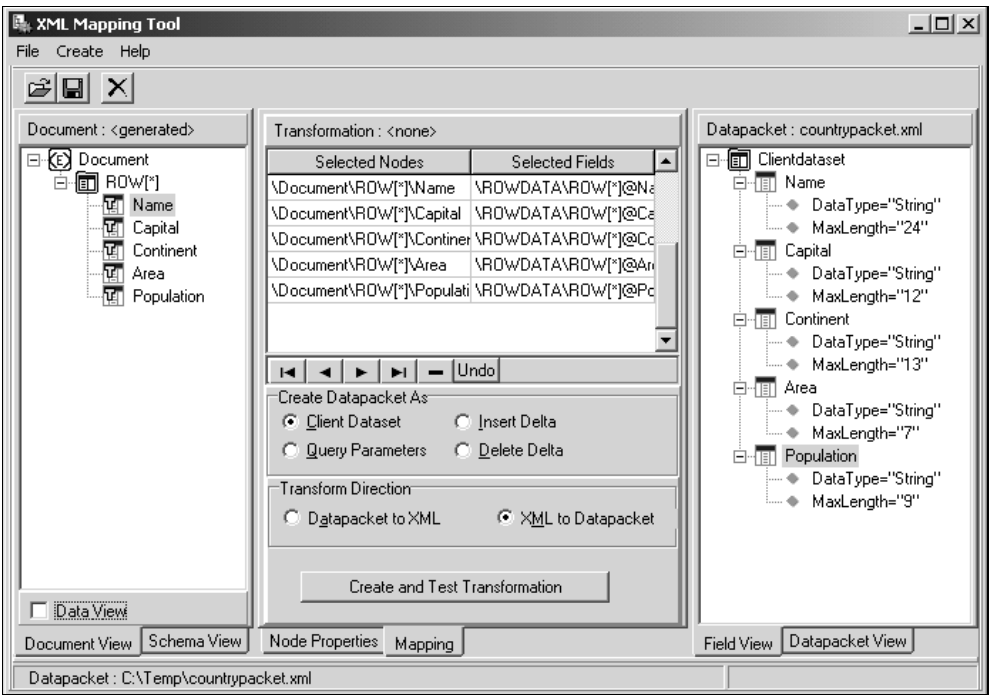


Рис. 14.2. Утилита XML Mapper позволяет создавать трансформационные файлы

Левая панель предназначена для работы с документом XML, который является исходным в цепочке преобразований. Выбор файла и его загрузка осуществляются стандартными элементами управления. Здесь на вкладке **Document View** можно работать с деревом узлов и элементов документа

XML. Здесь же выбираются элементы, предназначенные для преобразования. Вкладка **Schema View** служит для просмотра схемы документа.

Правая панель обеспечивает операции с файлом пакета данных. Вкладка **Field View** представляет структуру пакета данных в виде дерева полей. Вкладка **Datapacket View** содержит XML представление структуры пакета данных. (Ранее говорилось, что сохраненный во внешнем файле пакет данных клиентского набора данных имеет формат XML.) Выбор полей для связывания с элементами XML осуществляется на обеих вкладках.

Средняя панель отображает установленные связи между документом XML и пакетом данных и позволяет редактировать их. Для этого предназначена вкладка **Mapping**, в верхней части которой помещена таблица связей. А на вкладке **Node Properties** можно просмотреть свойства узла XML, выбранного в левой панели документа XML.

Выбор исходного файла

Утилита XML Mapper работает со следующими типами данных. Основными типами, для которых собственно и выполняется преобразование данных, являются:

- ❒ формат XML;
- ❒ формат пакета данных Delphi.

Информация о настроенных связях трансформации и вспомогательные данные представлены следующими типами данных (информация в них хранится в формате XML):

- ❒ трансформационный файл (XTR) хранит карту связей преобразования и описание обоих наборов данных (XML и пакет данных);
- ❒ файл репозитория (XRP);
- ❒ файл схемы XML (DTD, XDR, XSD) позволяет просмотреть структуру файла XML.

XML Mapper позволяет осуществлять трансформацию данных в обоих направлениях: из XML в пакет данных и обратно. Поэтому здесь можно открывать исходный файл как в левой, так и в правой панелях. Для этого пригодны стандартные средства Windows: главное меню, кнопку панели инструментов, всплывающее меню панели.

Кроме этого, пакет данных можно получить напрямую от удаленного сервера. Для этого предназначена команда **Connect To Remote Server** всплывающего меню утилиты. После ее выбора появляется панель (рис. 14.3), в которой необходимо выбрать нужный сервер (он должен быть зарегистрирован на компьютере) и набора данных этого сервера.

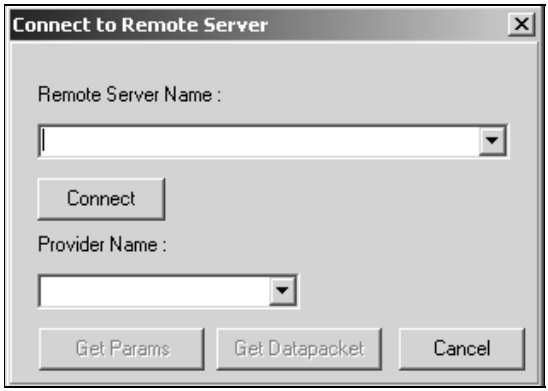


Рис. 14.3. Панель загрузки пакета данных с удаленного сервера

А при загрузке трансформационного файла вы получаете среду XML Mapper полностью готовой к работе — с данными XML и пакета данных и созданными связями (если, конечно, все это было сохранено).

Создание пакета данных и документа XML и сохранение преобразованных данных

В утилите XML Mapper для преобразования данных должен существовать как минимум файл-источник данных. Как уже говорилось, это может быть как файл XML, так и пакет данных. После выбора исходного файла с данными, подлежащими трансформации, необходимо открыть или создать файл-приемник данных.

Если файл-приемник существует, то его достаточно открыть в XML Mapper. Если же вы хотите создать новый файл, то после связывания элементов и полей необходимо вызвать команду из меню **Create**:

- **Datapacket from XML** — для создания пакет данных;
- **XML from Datapacket** — для создания файла XML.

При создании файла XML на основе существующего пакета вы можете дополнительно настроить создаваемые элементы. Для этого предназначено окно **Create Xml from ClientDataSet**, которое автоматически открывается при выборе команды **Create XML from Datapacket** всплывающего меню в правой панели утилиты. Здесь вы можете изменить имена элементов и корневого элемента (рис. 14.4).

А для создаваемого на основе документа XML пакета данных можно выбрать его тип. Для этого имеется группа радиокнопок **Craete Datapacket As** на вкладке **Mapping** центральной панели утилиты (см. рис.14.2).

После этого созданную структуру можно сохранить в файле при помощи обычных средств.

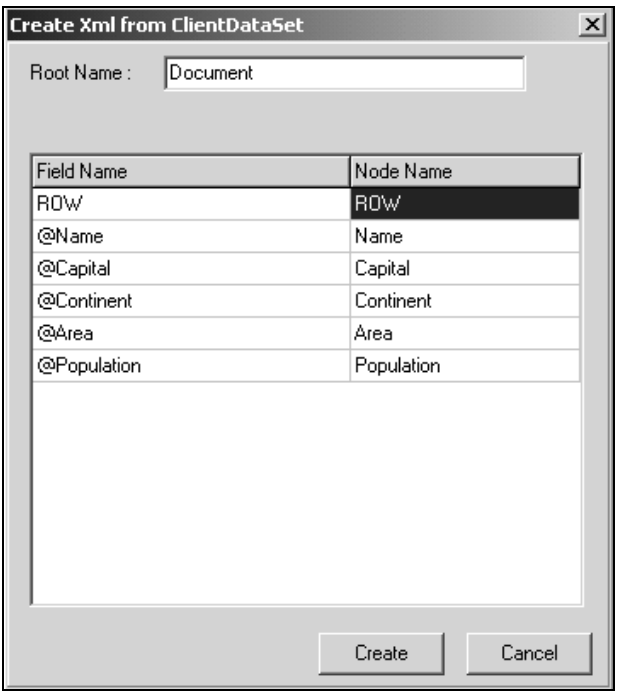


Рис. 14.4. Окно настройки элементов создаваемого документа XML

Примечание

Обратите внимание, что описываемые здесь команды создают только структуру данных, без переноса реальных данных. Собственно преобразование данных рассматривается далее.

Связывание элементов XML и полей пакета данных

Каким же образом выполняется связывание? Все очень просто. Вам необходимо лишь выбрать нужный элемент XML или поле пакета файла-источника данных в левой или правой панели и дважды щелкнуть на нем мышкой. В результате, в центральной панели в таблице на вкладке **Mapping**

появится новая запись, в которой будет фигурировать выбранный элемент или поле.

Дальше ваши действия зависят от того, существует или нет файл-приемник данных. Если нет — больше ничего делать не надо и можно переходить к следующему элементу или полю.

Если же файл-приемник существует, нужно аналогичным образом выбрать в его структуре требуемое поле или элемент и дважды щелкнуть на нем. В результате строка в таблице связей на вкладке **Mapping** окажется полностью заполненной и будет содержать связанные между собой элемент XML и поле пакета данных. Эта связь означает, что данные из этого элемента (поля) файла-источника будут перемещены в соответствующее поле (элемент) файла-приемника.

Примечание

Обратите внимание, что строки таблицы заполняются сверху вниз, без учета местоположения текущей записи.

После связывания всех требуемых полей и элементов XML таблица связывания будет содержать модель преобразования данных.

Создание трансформационного файла и преобразование данных

Когда таблица связей будет заполнена, можно приступать к созданию трансформационного файла. Для этого необходимо выбрать команду **Transformation** из меню **Create**. После этого в меню **File | Save** становится доступной команда **Transformation**. Выбрав ее, вы можете сохранить созданный трансформационный файл и затем использовать его при работе с компонентами трансформации.

Помимо создания трансформационного файла вы также можете выполнить и перенос данных между XML и пакетом данных. Для этого после выбора источника и приемника данных и настройки связей между полями и элементами XML необходимо нажать кнопку **Create and Test Transformation** на вкладке **Mapping** центральной панели. При этом направление преобразования данных задается в группе радиокнопок **Transform Direction**.

После этого выполняется собственно преобразование данных и его результат появляется в новом окне. Если приемником данных выступает документ XML, результат представлен в виде дерева XML (рис. 14.5). Для пакета данных результат отображается в виде таблицы.

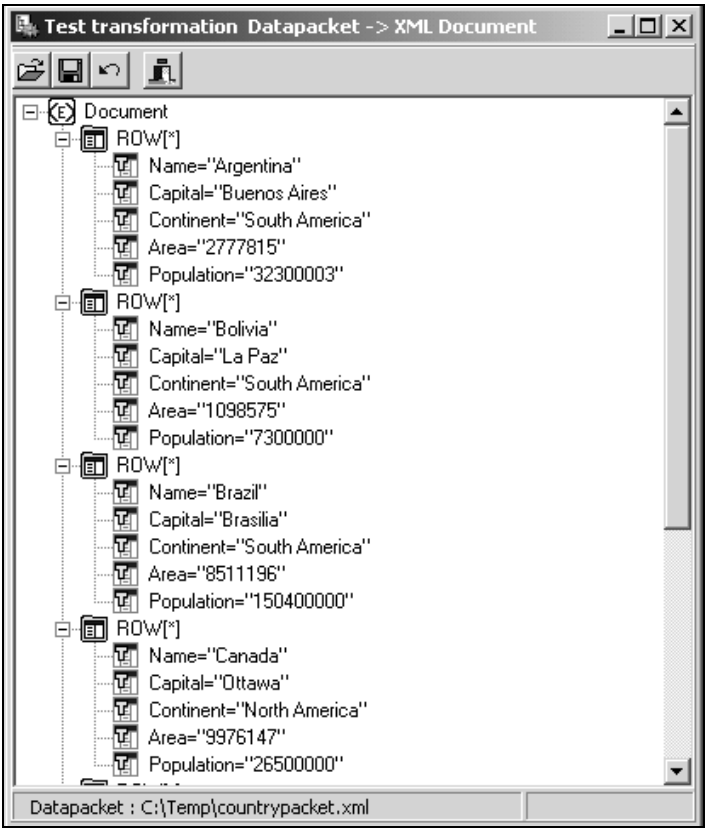


Рис. 14.5. Окно результата преобразования данных для документа XML

Резюме

Компоненты преобразования данных XML позволяют приложениям обмениваться данными XML. Для преобразования данных XML в формат пакета данных Delphi (который и позволяет работать с этими данными без особого труда) в распоряжении разработчиков имеется утилита XML Mapper.

ЧАСТЬ IV

Приложения баз данных

Глава 15

Архитектура приложений баз данных

Приложение баз данных, как следует уже из его названия, предназначено для взаимодействия с некоторым источником данных — базой данных. Взаимодействие подразумевает получение данных, их представление в определенном формате для просмотра пользователем, редактирование в соответствии с реализованными в программе бизнес-алгоритмами и возврат обработанных данных обратно в базу данных.

В качестве источника данных могут выступать как собственно базы данных, так и обычные файлы (текстовые, электронные таблицы и т. д.). Но здесь мы будем рассматривать приложения, работающие с базами данных.

Как известно, базы данных обслуживаются специальными программами — системами управления базами данных (СУБД), которые делятся на локальные, преимущественно однопользовательские, предназначенные для настольных приложений, и серверные (сетевые, часто удаленные), многопользовательские, функционирующие на выделенных компьютерах (серверах). Главный критерий такой классификации — объем базы данных и средняя нагрузка на СУБД.

Тем не менее, несмотря на разнообразие реализаций, общая архитектура приложения баз данных остается неизменной.

Само приложение включает механизм получения и отправки данных, механизм их внутреннего представления в том или ином виде, пользовательский интерфейс для отображения и редактирования данных и бизнес-логику для их обработки.

- Механизм получения и отправки данных — обеспечивает соединение с источником данных (часто опосредованно). Он должен "знать", куда ему обращаться и какой протокол обмена использовать для обеспечения двунаправленного потока данных.
- Механизм внутреннего представления данных — ядро приложения баз данных. Он обеспечивает хранение полученных данных в приложении и предоставляет их по запросу других частей приложения.

- Пользовательский интерфейс — обеспечивает просмотр и редактирование данных, а также управление данными и приложением в целом.
- Бизнес-логика приложения — набор реализованных в программе алгоритмов обработки данных.

Между приложением и собственно базой данных находится специальное программное обеспечение, связывающее программу и источник данных и управляющее процессом обмена данными. Это ПО может быть реализовано самыми разнообразными способами, в зависимости от объемов базы данных, решаемых системой задач, числа пользователей, способами соединения приложения и базы данных. Промежуточное ПО может быть реализовано в виде окружения приложения, без которого оно вообще не будет работать, как набор драйверов и динамических библиотек, к которым обращается приложение, может быть интегрировано в само приложение. Наконец, это может быть отдельный удаленный сервер, обслуживающий тысячи приложений.

Источник данных представляет собой хранилище данных (саму базу данных) и СУБД, управляющую данными, обеспечивающую целостность и непротиворечивость данных.

В этой и последующих главах третьей части мы подробно остановимся на способах разработки приложений баз данных в Delphi. При разнообразии способов реализации и обилии технических деталей, общая архитектура приложений баз данных в Delphi следует описанной общей схеме.

В Delphi реализовано достаточно большое число разнообразных технологий доступа к данным (они рассматриваются далее в этой книге). Но последовательность операций при конструировании приложений баз данных остается почти одинаковой. И употребляются по сути одни и те же компоненты, доработанные для применения с той или иной технологией доступа к данным.

В этой главе рассматриваются общие подходы к разработке приложений баз данных в Delphi, базовые классы и механизмы, которые не изменятся, выберете ли вы для вашего приложения Borland Database Engine (BDE), dbGo на основе Microsoft ActiveX Data Objects (ADO) или dbExpress.

Итак, в этой главе рассматриваются следующие вопросы.

- Структура приложения баз данных в Delphi.
- Базовые компоненты, необходимые при разработке приложений баз данных, и их взаимосвязь.
- Понятие набора данных, и его участие в основных механизмах приложения баз данных.
- Модуль данных.
- Программная реализация частей приложения баз данных.

Как работает приложение баз данных

В Репозитории Delphi отсутствует отдельный шаблон для приложения баз данных. Поэтому, как и любое другое приложение Delphi, приложение баз данных начинается с обычной формы. Безусловно, это оправданный подход, т. к. приложение баз данных имеет пользовательский интерфейс. И этот интерфейс создается на основе стандартных и специализированных визуальных компонентов на обычных формах.

Визуальные компоненты отображения данных расположены на странице **Data Controls** Палитры инструментов. В большинстве они представляют собой модификации стандартных элементов управления, приспособленные для работы с набором данных *(см. главу 18)*.

Приложение может содержать произвольное число форм и задействовать любой интерфейс (MDI или SDI). Обычно одна форма отвечает за выполнение группы однородных операций, объединенных общим назначением.

В основе любого приложения баз данных лежат *наборы данных*, которые представляют собой группы записей (в виде таблиц в памяти), переданных из базы данных в приложение для просмотра и редактирования. Каждый набор данных инкапсулирован в специальном компоненте доступа к данным. В VCL Delphi реализован набор базовых классов, поддерживающих функциональность наборов данных, и практически идентичные по составу наборы дочерних компонентов для технологий доступа к данным. Их общий предок — класс `TDataSet`. (Подробно наборы данных рассмотрены *в главе 16*.)

Для обеспечения связи набора данных с визуальными компонентами отображения данных есть специальный компонент `TDataSource`. Его роль заключается в управлении потоками данных между набором данных и связанными с ним компонентами отображения. Этот компонент обеспечивает передачу данных в визуальные компоненты и возврат результатов редактирования в набор данных, отвечает за изменение состояния визуальных компонентов при изменении состояния набора данных, передает сигналы управления от пользователя (визуальных компонентов) в набор данных. Компонент `TDataSource` расположен в категории **Data Access** Палитры инструментов.

Таким образом, базовый механизм доступа к данным создается триадой компонентов:

- компоненты, инкапсулирующие набор данных (потомки класса `TDataSet`);
- компоненты `TDataSource`;
- визуальные компоненты отображения данных.

Рассмотрим схему взаимодействия этих компонентов в приложении баз данных (рис. 15.1).

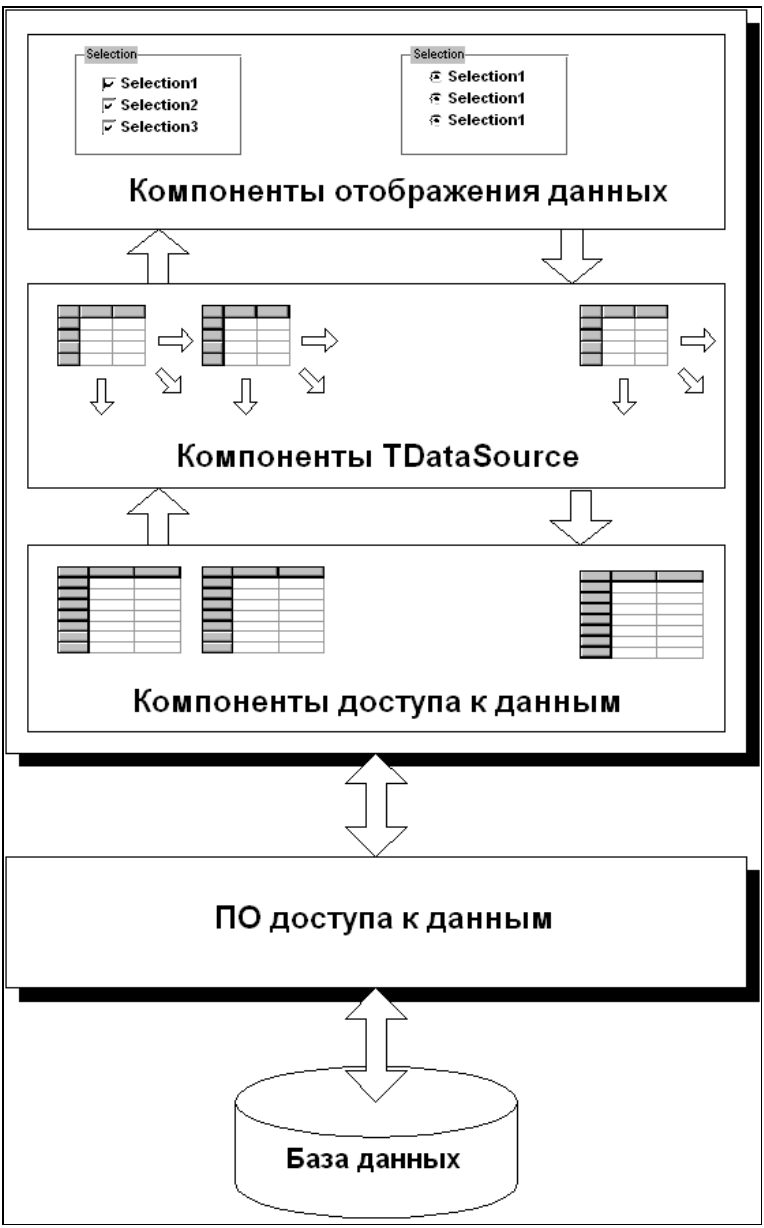


Рис. 15.1. Механизм доступа к данным приложения баз данных

В приложении с источником данных или промежуточным программным обеспечением взаимодействует компонент доступа к данным (см. рис. 15.1), который инкапсулирует набор данных и обращается к функциям соответствующей технологии доступа к данным для выполнения различных операций. Компонент доступа к данным представляет собой "образ" таблицы базы данных в приложении. Общее число таких компонентов в приложении не ограничено.

С каждым компонентом доступа к данным может быть связан как минимум один компонент `TDataSource`. В его обязанности входит соединение набора данных с визуальными компонентами отображения данных. Компонент `TDataSource` обеспечивает передачу в эти компоненты текущих значений полей из набора данных и возврат в него сделанных изменений.

Еще одна функция компонента `TDataSource` заключается в синхронизации поведения компонентов отображения данных с состоянием набора данных. Например, если набор данных неактивен, то компонент `TDataSource` обеспечивает удаление данных из компонентов отображения данных и их перевод в неактивное состояние. Или, если набор данных работает в режиме "только для чтения", то компонент `TDataSource` обязан передать в компоненты отображения данных запрещение на изменение данных.

С одним компонентом `TDataSource` может быть связано несколько визуальных компонентов отображения данных. Эти компоненты представляют собой модифицированные элементы управления, которые предназначены для показа информации из наборов данных.

При открытии набора данных компонент обеспечивает передачу в набор данных записей из требуемой таблицы БД. Курсор набора данных устанавливается на первую запись. Компонент `TDataSource` организует передачу в компоненты отображения данных значений необходимых полей из текущей записи. При перемещении по записям набора данных текущие значения полей в компонентах отображения данных автоматически обновляются.

Пользователь при помощи компонентов отображения данных может просматривать и редактировать данные. Измененные значения сразу же передаются из элемента управления в набор данных при помощи компонента `TDataSource`. Затем изменения могут быть переданы в базу данных или отменены.

Теперь, имея общее представление о работе приложения баз данных, перейдем к поэтапному рассмотрению процесса создания такого приложения.

Модуль данных

Для размещения компонентов доступа к данным в приложении баз данных желательно использовать специальную "форму" — модуль данных (класс

TDataModule). Обратите внимание, что модуль данных не имеет ничего общего с обычной формой приложения, ведь его непосредственным предком является класс TComponent. В модуле данных можно размещать только невизуальные компоненты. Модуль данных доступен разработчику, как и любой другой модуль проекта, на этапе разработки (рис. 15.2). Пользователь приложения не может увидеть модуль данных во время выполнения.

Для создания модуля данных можно воспользоваться Репозиторием объектов Delphi.

Как уже говорилось, модуль данных имеет мало общего со стандартной формой хотя бы потому, что класс TDataModule происходит непосредственно от класса TComponent. У него почти полностью отсутствуют свойства и методы-обработчики событий, ведь от платформы для других невизуальных компонентов почти ничего не требуется, хотя потомки модуля данных, работающие в распределенных приложениях, выполняют весьма важную работу.

При помощи управляющих кнопок можно задавать между элементами диаграммы отношения синхронного просмотра и главный — подчиненный. При этом выполняется автоматическая настройка свойств соответствующих компонентов.

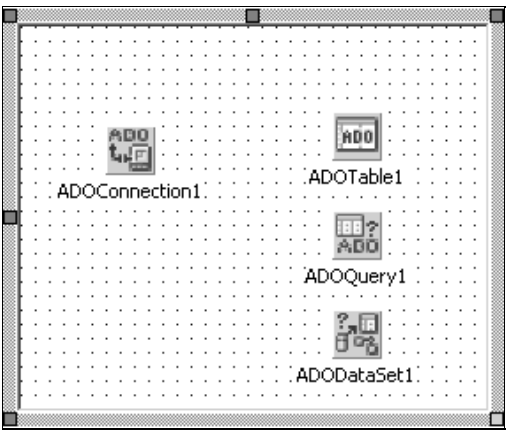


Рис. 15.2. Модуль данных

Для обращения к компонентам доступа к данным, расположенным в модуле данных, из других модулей проекта необходимо включить имя модуля в секцию uses:

```
unit InterfaceModule;
...
implementation
```

```
uses DataModule;
...
DataModule.Table1.Open;
...
```

Преимуществом размещения компонентов доступа к данным в модуле данных является то, что изменение значения любого свойства проявится сразу же во всех обычных модулях, к которым он подключен. Кроме этого, все обработчики событий этих компонентов, т. е. вся логика работы с данными приложения, собраны в одном месте, что тоже весьма удобно.

Подключение набора данных

Компонент доступа к данным является основой приложения баз данных. Используя выбранную таблицу БД, он создает набор данных и позволяет эффективно управлять им. В процессе работы такой компонент тесно взаимодействует с функциями соответствующей технологии доступа к данным. Обычно такой доступ осуществляется через совокупность интерфейсов. Все компоненты доступа к данным являются невизуальными.

Для создания нового проекта достаточно выбрать команду **New | VCL Forms Application — Delphi for Win32** из меню **File** или воспользоваться Репозиторием объектов, который открывается командой **New | Other** из меню **File**.

Примечание

Здесь рассматривается простейший вариант создания приложения. В реальных проектах для размещения компонентов доступа к данным следует использовать модуль данных.

Затем на форму нового проекта необходимо перенести компонент, инкапсулирующий набор данных, и выполнить следующие действия. Последовательность действий рассмотрим для компонента, инкапсулирующего функции таблицы *(см. главу 16)*.

1. Подключить компонент к базе данных. Для этого, в зависимости от конкретной технологии, применяется или специальный компонент, устанавливающий соединение, или прямое обращение к драйверу, интерфейсу или динамической библиотеке. Детально способы соединения рассматриваются в *части V*.
2. Подключить к компоненту таблицу БД. Для этого в Инспекторе объектов есть свойство `TableName`. После выполнения действий первого этапа в списке этого свойства должны появиться имена всех доступных в подключенной базе данных таблиц. После выбора имени таблицы в свойстве `TableName` компонент оказывается связанным с ней.

3. Переименовать компонент. Это необязательное действие. Тем не менее, в любых случаях желательно присваивать компонентам доступа к данным осмысленные имена, соответствующие названиям подключенных таблиц. Обычно название компонента копирует название таблицы (например, `Orders` или `OrdTable` или `tblOrders`).
4. Активизировать связь между компонентом и таблицей БД. Для этого служит свойство `Active`. Если в Инспекторе объектов присвоить этому свойству значение `True`, то связь активизируется. Эту операцию можно выполнить и в исходном коде приложения. Также существует метод `Open`, который открывает набор данных, и метод `Close`, закрывающий его.

В качестве примера попробуем создать простейшее приложение баз данных, работающее с таблицей COUNTRY.DB из стандартной демонстрационной базы данных DBDEMOS через драйвер процессора Borland Database Engine. При желании можно выбрать любую другую технологию доступа к данным, заменив табличный компонент. Подробнее о технологиях доступа к данным *см. часть V.*

На форму нового проекта необходимо перенести компонент `TTable` из категории **BDE** Палитры инструментов. Свойство `DatabaseName` должно ссылаться на псевдоним DBDEMOS, который создается автоматически при установке Delphi, его можно выбрать из списка свойства `DatabaseName`. Для свойства `TableName` необходимо задать имя таблицы `'COUNTRY.DB'`. Его также можно выбрать из списка. Двойной щелчок на свойстве `Active` в Инспекторе объектов присваивает ему значение `True`. После этого связь компонента с таблицей активизируется. Свойство `Name` имеет значение `'CountryTable'`.

Открытие и закрытие набора данных можно предусмотреть как реакцию на действия пользователя или возникновение события. Чаще всего набор данных должен открываться при первом показе формы и закрываться при ее закрытии. Листинг 15.1 иллюстрирует секци. `Implementation` главного модуля проекта DemoDBApp.

Листинг 15.1. Секция `implementation` главного модуля проекта DemoDBApp

```
implementation
{$R *.DFM}
procedure TForm1.FormShow(Sender: TObject);
begin
 try
  CountryTable.Open;
 except
```

```
  ShowMessage('Table open error');
 end;
end;
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 CountryTable.Close;
end;
end.
```

При открытии формы выполняется метод-обработчик `FormShow`. В нем набор данных открывается при помощи метода `Open`. Обратите внимание на конструкцию `try ... except`, которая обеспечивает корректное завершение при возникновении исключительных ситуаций.

Так как ошибки в работе приложений баз данных могут привести к серьезным последствиям (потеря или искажение данных), то защитный код должен присутствовать во всех критических местах.

В методе-обработчике `FormClose`, который вызывается при закрытии формы, набор данных закрывается методом `Close`.

Примечание

В принципе, для выполнения рассмотренных операций можно воспользоваться и свойством `Active`. Однако реальные операции выполняют указанные методы. Поэтому использование свойства является лишним этапом, да и с точки зрения ООП все действия должны выполнять методы объекта, а свойства служат только для представления значений. Свойство `Active` сигнализирует о состоянии набора данных.

Настройка компонента *TDataSource*

На втором этапе разработки приложения баз данных необходимо перенести на форму и настроить компонент `TDataSource`. Он обеспечивает взаимодействие набора данных с компонентами отображения данных. Чаще всего одному набору данных соответствует один компонент `TDataSource`, хотя их может быть несколько.

Для настройки свойств компонента необходимо выполнить следующие действия.

1. Связать набор данных и компонент `TDataSource`. Для этого предусмотрено свойство `DataSet` компонента `TDataSource`, доступное через Инспектор объектов. Это указатель на экземпляр компонента доступа к данным.

В списке этого свойства в Инспекторе объектов перечислены все доступные компоненты наборов данных.

2. Переименовать компонент. Это необязательное действие. Тем не менее, желательно присваивать компонентам осмысленные имена, соответствующие названиям связанных наборов данных. Обычно название компонента является комбинацией имен набора данных (например, `OrdSource` или `dsOrders`).

В приложении DemoDBApp компонент `CountrySource` связан с компонентом `CountryTable`. Для этого свойство `DataSet` имеет значение `CountryTable`.

Примечание

Компонент `TDataSource` можно подключить не только к набору данных из той же формы, но и любой другой, модуль которой указан в секции `uses`.

Компонент `TDataSource` имеет ряд полезных свойств и методов.

Итак, связывание с компонентом набора данных выполняет свойство

```
property DataSet: TDataSet;
```

а определить текущее состояние набора данных можно, использовав свойство

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead,
dsInternalCalc);
property State: TDataSetState;
```

При помощи свойства

```
property Enabled: Boolean;
```

можно включить или отключить все связанные визуальные компоненты. При значении `False` ни один связанный компонент отображения данных не будет работать.

Свойство

```
property AutoEdit: Boolean;
```

при значении `True` всегда будет переводить набор данных в режим редактирования при получении фокуса одним из связанных визуальных компонентов.

Аналогично, метод

```
procedure Edit;
```

переводит связанный набор данных в режим редактирования.

Метод

```
function IsLinkedTo(DataSet: TDataSet): Boolean;
```

возвращает значение `True`, если компонент, указанный в параметре `DataSet`, действительно связан с данным компонентом `TDataSource`.

Метод-обработчик

```
type TDataChangeEvent = procedure(Sender: TObject; Field: TField) of
object;
property OnDataChange: TDataChangeEvent;
```

вызывается при редактировании данных в одном из связанных визуальных компонентов.

Метод-обработчик

```
property OnUpdateData: TNotifyEvent;
```

вызывается перед сохранением изменений в базе данных.

Метод обработчик

```
property OnStateChange: TNotifyEvent;
```

вызывается при изменении состояния связанного набора данных *(см. главу 16)*.

Отображение данных

На третьем этапе создания приложения баз данных необходимо разработать пользовательский интерфейс на основе компонентов отображения данных. Эти компоненты предназначены специально для решения задач просмотра и редактирования данных. Внешне большинство этих компонентов ничем не отличаются от стандартных элементов управления. Более того, многие из компонентов отображения являются наследниками стандартных компонентов — элементов управления.

Компоненты отображения данных должны быть связаны с компонентом `TDataSource` и через него с компонентом набора данных. Для этого используется их свойство `DataSource`. Оно присутствует во всех компонентах отображения данных.

Большинство компонентов предназначены для представления данных из одного-единственного поля. В таких компонентах имеется еще одно свойство `DataField`, которое определяет поле связанного набора данных, отображаемое в компоненте.

Особое значение для приложений баз данных играет компонент `TDBGrid`, который представляет данные в виде таблицы. В столбцах таблицы размещаются поля набора данных, а в строках — записи. Для этого компонента нет смысла определять конкретное поле, но можно задать настраиваемый набор колонок, а для каждой из них определить поле набора данных. (Подробнее о визуальных компонентах отображения данных *см. главу 19.*)

Таким образом, для каждого визуального компонента отображения данных необходимо выполнить следующие операции.

1. Связать компонент отображения данных и компонент `TDataSource`. Для этого предназначено свойство `DataSource`, которое должно указывать на экземпляр требуемого компонента `TDataSource`. Один компонент отображения данных можно связать только с одним компонентом `TDataSource`. Необходимый компонент можно выбрать в списке свойства в Инспекторе объектов.
2. Задать поле данных. Для этого есть свойство `DataField` типа `TFields`. В нем необходимо указать имя поля связанного набора данных. После задания свойства `DataSource` поле можно выбрать из списка. Этот этап применяется только для компонентов, отображающих единственное поле.

Отдельное место среди компонентов отображения данных занимает компонент `TDBNavigator`. Он предназначен для перемещения по записям набора данных.

В приложении DemoDBApp задействованы компоненты `TDBGrid`, `TDBNavigator` и `TDBEdit` (рис. 15.3).

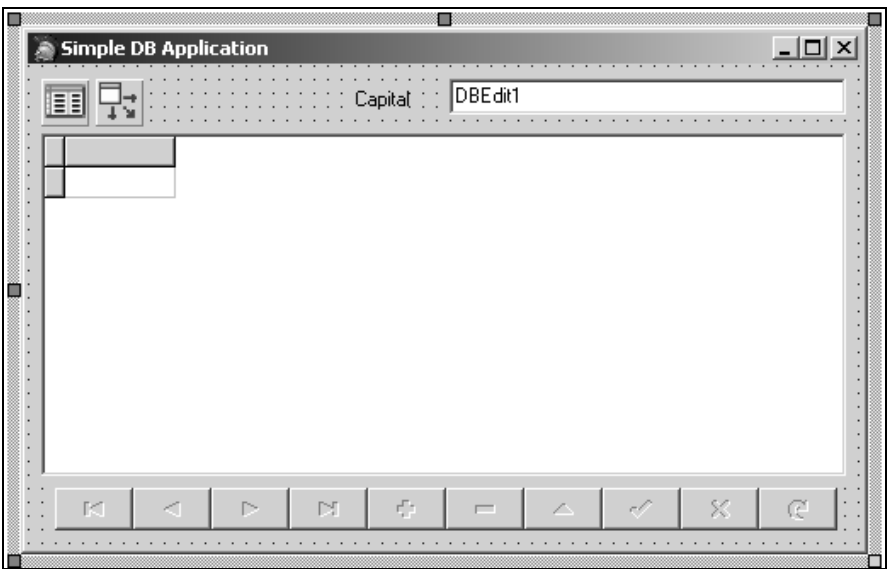


Рис. 15.3. Главная форма приложения DemoDBApp

Все три компонента отображения данных связаны с компонентом `CountrySource` типа `TDataSource` при помощи свойства `DataSource`.

Компонент `TDBEdit` отображает данные из поля `Capital` (столица государства) и позволяет редактировать их.

Компонент `TDBGrid` показывает набор данных целиком, данные в ячейках можно редактировать.

Компонент `TDBNavigator` позволяет перемещаться по записям набора данных `CountryTable`. При этом результат заметен во всех подключенных к набору данных компонентах отображения данных.

Резюме

Приложения баз данных могут получать доступ к источникам данных при помощи разнообразных технологий доступа, многие из которых доступны и в приложениях Delphi. Тем не менее, любое приложение баз данных в Delphi имеет стандартное ядро, структура которого определена архитектурой приложения баз данных.

Набор базовых компонентов и способов разработки является единой основой, на которой базируются технологии доступа к данным. Это позволило унифицировать процесс разработки приложений баз данных.

В основе процесса разработки лежит триада компонентов:

- невизуальные компоненты набора данных;
- невизуальные компоненты `TDataSource`;
- визуальные компоненты отображения данных.

Глава 16

Набор данных

Любое приложение баз данных должно уметь выполнять как минимум две операции. Во-первых, иметь информацию о местонахождении базы данных, подключаться к ней и считывать имеющуюся в таблицах БД информацию. Эта функция в значительной степени зависит от реализации конкретной технологии доступа к данным.

Во-вторых, обеспечивать представление и редактирование полученных данных. Множество записей одной или нескольких таблиц, переданные в приложение в результате активизации компонента доступа к данным, будем называть *набором данных*. Понятно, что в объектно-ориентированной среде для представления какой-либо группы записей приложение должно использовать возможности некоторого класса, который должен инкапсулировать набор данных и обладать методами для управления записями и полями.

Таким образом, сам набор данных и класс набора данных является той осью, вокруг которой вращается любая деятельность приложения баз данных.

Пользователь просматривает на экране данные — это результат использования набора данных.

Пользователь решил изменить какое-то число — он изменит содержимое ячейки набора данных.

При закрытии приложение сохраняет все изменения — это набор данных передается в базу данных для сохранения.

При этом, используя одни базовые функции для обслуживания набора данных, компоненты должны обеспечивать доступ к данным в рамках различных технологий. Поэтому неудивительно, что разработчики VCL уделили особое внимание созданию максимально эффективной иерархии классов, обеспечивающих работу с наборами данных (рис. 16.1).

Класс `TDataSet` является базовым классом иерархии, он инкапсулирует абстрактный набор данных и реализует максимально общие методы работы с ним. В него можно передать записи из таблицы базы данных или строки из

обычного текстового файла — набор данных будет функционировать одинаково хорошо.

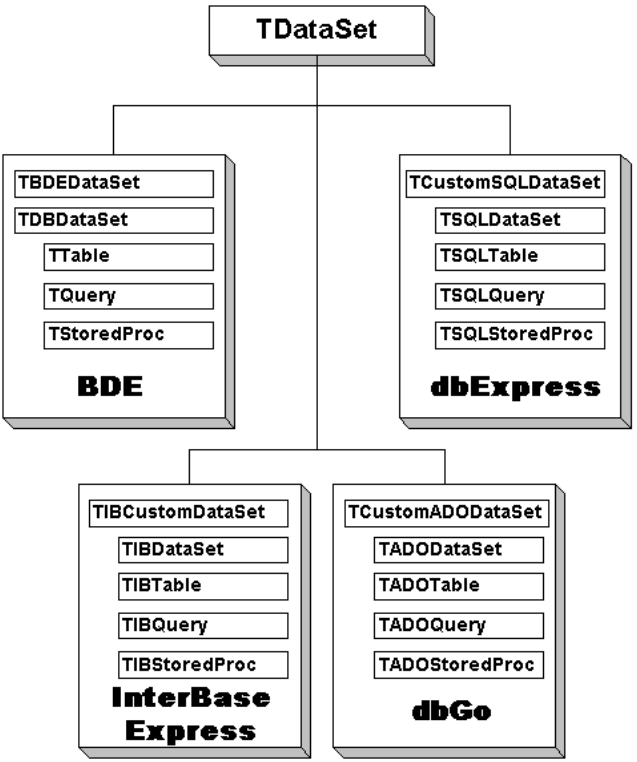


Рис. 16.1. Иерархия классов, обеспечивающих функционирование набора данных

На основе базового класса реализованы специальные компоненты VCL для различных технологий доступа к данным, которые позволяют разработчику конструировать приложения баз данных, с помощью одинаковых приемов и свойств.

В этой главе рассматриваются следующие вопросы.

- Набор данных, инкапсулированный в классе `TDataSet`.
- Что такое состояния набора данных.
- Индексы, поля, параметры.
- Прототипы компонентов для работы с таблицами, запросами и хранимым процедурами.
- Основные механизмы набора данных, реализованные в классе `TDataSet`.

Абстрактный набор данных

В основе иерархии классов, обеспечивающих функционирование наборов данных в приложениях баз данных Delphi, лежит класс `TDataSet`. Хотя он почти не содержит методов, реально обеспечивающих работоспособность основных механизмов набора данных, тем не менее его значение трудно переоценить.

Этот класс задает структурную основу функционирования набора данных. Другими словами, это скелет набора данных, к методам которого необходимо лишь добавить требуемые вызовы соответствующих функций реальных технологий.

При решении наиболее распространенных задач программирования в процессе создания приложений баз данных класс `TDataSet` не нужен. Тем не менее, знание основных принципов работы набора данных всегда полезно. Кроме этого, класс `TDataSet` может быть основой для создания собственных пользовательских компонентов. Поэтому рассмотрим основные механизмы, реализованные в наборе данных.

Набор данных открывается и закрывается свойством

```
property Active: Boolean;
```

которому соответственно необходимо присвоить значение `True` или `False`.

Аналогичные действия выполняют методы

```
procedure Open;
procedure Close;
```

После открытия набора данных можно перемещаться по его записям.

На одну запись вперед и назад перемещают курсор соответственно методы

```
procedure Next;
procedure Prior;
```

На первую и последнюю запись можно попасть, используя соответственно методы

```
procedure First;
procedure Last;
```

Признаком того, что достигнута последняя запись набора, является свойство

```
property Eof: Boolean;
```

которое в этом случае имеет значение `True`.

Аналогичную функцию для первой записи выполняет свойство

```
property Bof: Boolean;
```

Перемещение вперед и назад на заданное число записей выполняет метод

```
function MoveBy(Distance: Integer): Integer;
```

Параметр `Distance` определяет число записей. Если он отрицательный — перемещение осуществляется к началу набора данных, иначе — к концу.

Для ускоренного перемещения по набору данных можно отключить все связанные компоненты отображения данных. Это делается методом

```
procedure DisableControls;
```

Обратная операция выполняется методом

```
procedure EnableControls;
```

Общее число записей набора данных возвращает свойство

```
property RecordCount: Integer;
```

однако применять его нужно аккуратно, т. к. каждое обращение к этому свойству приводит к обновлению набора данных, что может вызвать проблемы для больших таблиц или сложных запросов. Если вам нужно определить, не является ли набор данных пустым (часто встречающаяся операция), можно выбрать метод

```
function IsEmpty: Boolean;
```

который возвращает `True`, если набор данных пуст, или уже упоминавшиеся свойства

```
...
if MyTable.Bof and MyTable.Eof
 then ShowMessage('DataSet is empty');
...
```

Номер текущей записи позволяет узнать свойство

```
property RecNo: Integer;
```

Размер записи в байтах возвращает свойство

```
property RecordSize: Word;
```

Каждая запись набора данных представляет собой совокупность значений полей таблицы. В зависимости от типа компонента и его настройки число полей в наборе данных может изменяться. И совсем не обязательно набор данных должен содержать все поля таблицы базы данных.

Совокупность полей набора данных инкапсулирует свойство

```
property Fields: TFields;
```

а все необходимые параметры полей содержатся в свойстве

```
property FieldDefs: TFieldDefs;
```

Общее число полей набора данных возвращает свойство

```
property FieldCount: Integer;
```

а общее число полей типа BLOB содержится в свойстве

```
property BlobFieldCount: Integer;
```

Доступ к значениям полей текущей записи предоставляет свойство

```
property FieldValues[const FieldName: string]: Variant; default;
```

в параметре `FieldName` задается имя поля.

В процессе программирования разработчик очень часто обращается к полям набора данных. Если структура полей набора данных жестко задана и не изменяется, это можно сделать так:

```
for i := 0 to MyTable.FieldCount – 1 do
 MyTable.Fields[i].DiplayFormat := '#.###';
```

Иначе, если порядок следования полей и их состав меняются, можно использовать метод

```
function FieldByName(const FieldName: string): TField;
```

И делается это так:

```
MyTable.FieldByName('VENDORNO').AsInteger := 1234;
```

Имя поля, передаваемое в параметре `FieldName`, нечувствительно к регистру символов.

Метод

```
procedure GetFieldNames(List: TStrings);
```

вернет в параметр `List` полный список имен полей набора данных.

Более подробная информация о полях и способах работы с ними содержится в *главе 17*.

Класс `TDataSet` содержит ряд свойств и методов, которые обеспечивают редактирование набора данных.

Но сначала бывает полезно поинтересоваться, можно ли редактировать набор данных вообще. Это можно сделать при помощи свойства

```
property CanModify: Boolean;
```

которое принимает значение `True` для редактируемых наборов.

Перед началом редактирования набор данных нужно перевести в режим редактирования методом

```
procedure Edit;
```

Для сохранения сделанных изменений применяется метод

```
procedure Post; virtual;
```

Разработчик может вызывать его самостоятельно, или же `Post` вызывается самим набором данных при переходе на другую запись.

При необходимости все сделанные после последнего вызова `Post` изменения можно отменить методом

```
procedure Cancel; virtual;
```

Новая пустая запись добавляется в конец набора данных методом

```
procedure Append;
```

Новая пустая запись добавляется на место текущей методом

```
procedure Insert;
```

а текущая запись и все нижеследующие смещаются на одну позицию вниз.

Внимание

При использовании методов `Append` и `Insert` набор данных переходит в режим редактирования самостоятельно.

Дополнительно, у вас есть возможность добавить или вставить новую запись уже с заполненными полями. Для этого применяются методы

```
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
```

А делается это примерно так:

```
MyTable.AppendRecord([2345, 'New customer', '+7(812)4569012', 0, '']);
```

После вызова этих методов и их завершения набор данных автоматически возвращается в состояние просмотра.

А для существующей записи аналогичным образом можно заполнить все поля методом

```
procedure SetFields(const Values: array of const);
```

Текущая запись удаляется методом

```
procedure Delete;
```

При этом набор данных не выдает никаких предупреждений, а просто делает это.

Очистить содержимое всех полей текущей записи может метод

```
procedure ClearFields;
```

Обратите внимание, что поля становятся пустыми (NULL), а не сбрасываются в нулевое значение.

О том, редактировалась ли текущая запись, сообщает свойство

```
property Modified: Boolean;
```

если оно имеет значение True.

Набор данных можно обновить, не закрывая и не открывая его снова. Для этого применяется метод

```
procedure Refresh;
```

Однако он сработает только для таблиц и тех запросов, которые нельзя редактировать.

В каждый момент времени набор данных находится в определенном состоянии (о состояниях см. далее в этой главе). Свойство

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead,
dsInternalCalc, dsOpening);
property State: TDataSetState;
```

дает информацию о текущем состоянии набора.

Методы-обработчики класса TDataSet предоставляют разработчику широчайшие возможности по отслеживанию событий, происходящих с набором данных.

По паре методов-обработчиков (до и после события) предусмотрено для следующих событий в наборе данных:

- открытие и закрытие набора данных;
- переход в режим редактирования;
- переход в режим вставки новой записи;
- сохранение сделанных изменений;
- отмена сделанных изменений;
- перемещение по записям набора данных;
- обновление набора данных.

Обратите внимание, что помимо методов-обработчиков режима вставки существует еще один метод

```
property OnNewRecord: TDataSetNotifyEvent;
```

который вызывается непосредственно при вставке или добавлении записи.

Дополнительно к этому могут вызываться методы-обработчики возникающих ошибок. Они предусмотрены для ошибок удаления, редактирования и сохранения изменений.

Метод-обработчик

```
property OnCalcFields: TDataSetNotifyEvent;
```

очень важен для задания значений вычисляемых полей. Он вызывается для каждой записи, которая отображается в визуальных компонентах, связанных с набором данных каждый раз, когда необходимо перерисовать значения полей в визуальных компонентах.

Если в методе-обработчике `OnCalcFields` производятся слишком сложные вычисления, частота его вызовов может выть уменьшена за счет свойства

```
property AutoCalcFields: Boolean;
```

По умолчанию оно равно `True`, и расчет вычисляемых полей производится при каждой перерисовке. При значении `False` метод-обработчик `OnCalcFields` вызывается только при открытии, переходе в состояние редактирования и обновлении набора данных.

Все перечисленные обработчики имеют одинаковый тип

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
```

И метод-обработчик

```
type TFilterRecordEvent = procedure(DataSet: TDataSet; var Accept:
Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

вызывается для каждой записи набора данных при свойстве `Filtered = True`.

Помимо перечисленных, класс `TDataSet` содержит еще множество свойств и методов, которые обеспечивают работоспособность многих функций, полезных в практическом программировании приложений баз данных. Подробно они рассмотрены в *главе 18*.

Стандартные компоненты

Внимательный читатель заметил, что на рис. 16.1 набор компонентов для каждой из представленных технологий доступа к данным примерно одинаков. Везде есть компонент, инкапсулирующий табличные функции, компонент запроса SQL и компонент хранимой процедуры. И хотя все они имеют разных ближайших предков, тем не менее функциональность подобных компонентов в разных технологиях почти одинакова.

Поэтому имеет смысл рассмотреть общие для компонентов свойства и методы, представив, что существуют некие виртуальные общие предки для таблицы, запроса и хранимой процедуры.

Примечание

Некоторые из описываемых далее свойств и методов присутствуют не в каждой реализации компонентов.

Компонент таблицы

Компонент таблицы обеспечивает доступ к таблице базы данных целиком, создавая набор данных, структура полей которого полностью повторяет таблицу БД. За счет этого компонент прост в настройке и обладает многими дополнительными функциями, которые обеспечивает применение табличных индексов.

Но в практике программирования работа с таблицами целиком встречается не так часто. А при работе с серверами баз данных промежуточное ПО технологий доступа к данным все равно транслирует запрос на получение табличного набора данных в простейший запрос SQL, например:

```
SELECT * FROM Orders
```

В такой ситуации табличные компоненты менее эффективны по сравнению с запросами.

После соединения с источником данных (процесс подключения для каждой технологии подробно рассматривается в *части V*) необходимо задать имя таблицы в свойстве

```
property TableName: String;
```

Иногда в свойстве `TableType` дополнительно задается тип таблицы.

Если соединение с источником данных настроено правильно, имя таблицы можно выбрать из выпадающего списка свойства `TableName`.

Преимуществом табличного компонента являются индексы, которые ускоряют работу с таблицей. Все индексы, созданные в базе данных для таблицы, автоматически загружаются в компонент. Их параметры доступны через свойство

```
property IndexDefs: TIndexDefs;
```

подробно класс `TIndexDefs` рассматривается далее в этой главе.

При работе с компонентом разработчик имеет возможность управлять индексами.

Существующий индекс можно выбрать в Инспекторе объектов в списке свойства

```
property IndexName: String;
```

или использовать свойство

```
property IndexFieldNames: String;
```

в котором можно задать произвольное сочетание имен индексированных полей таблицы. Имена полей разделяются точкой с запятой. Таким образом, при помощи свойства `IndexFieldNames` можно создавать составные индексы.

Свойства `IndexName` и `IndexFieldNames` одновременно недопустимы.

Число полей в текущем индексе табличного компонента возвращает свойство

```
property IndexFieldCount: Integer;
```

А свойство

```
property IndexFields: [Index: Integer]: TField;
```

представляет собой индексированный список полей, входящих в текущий индекс:

```
...
for i := 0 to MyTable.IndexFieldCount - 1 do
 MyTable.IndexFields[i].Enabled := False;
...
```

Для выполнения операций с таблицами и индексами целиком в табличных компонентах реализовано несколько методов.

Метод

```
procedure CreateTable;
```

создает новую таблицу в базе данных, используя заданное имя и описание полей и индексов из свойств `TFieldDefs` и `TIndexDefs`. Если таблица с таким именем уже имеется в базе данных, то она будет уничтожена и создана заново с новой структурой и данными.

Метод

```
procedure EmptyTable;
```

удаляет из набора данных и таблицы базы данных все записи.

Метод

```
procedure DeleteTable;
```

уничтожает таблицу базы данных, связанную с компонентом. Набор данных должен быть закрыт.

Метод

```
type
  TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive,
ixExpression, ixNonMaintained);
  TIndexOptions = set of TIndexOption;
procedure AddIndex(const Name, Fields: String; Options: TIndexOptions,
const DescFields: String='');
```

добавляет к таблице БД новый индекс. Параметр `Name` задает имя индекса. В параметре `Fields` через точку с запятой определяются имена полей, входящих в индекс. Параметр `DescFields` задает описание индекса из констант, объявленных в типе `TIndexOption`.

Метод

```
procedure DeleteIndex(const Name: string);
```

уничтожает индекс.

Кроме этого, табличные компоненты содержат свойства и методы, описываемые в *главе 18*.

Компонент запроса

Компонент запроса предназначен для создания запроса SQL, подготовки его параметров, передачи запроса на сервер БД и представления результата запроса в наборе данных. При этом набор данных может быть редактируемым или нет.

Любой компонент запроса, каждая строка набора данных которого однозначно связывается с одной строкой таблицы БД, может редактироваться. Например, запрос

```
SELECT * FROM Country
```

редактировать можно. Если же приведенное правило не выполняется, то набор данных доступен только для просмотра, и конечно, возможности компонентов здесь ни при чем. Куда, к примеру, записывать результаты редактирования записей следующего запроса:

```
SELECT CustNo, SUM(AmountPaid)
FROM Orders
GROUP BY CustNo
```

Ведь в таком запросе каждая запись — есть результат суммирования неизвестного заранее числа других записей.

Тем не менее компоненты запросов предоставляют разработчику мощный и гибкий механизм работы с данными. С помощью компонентов запросов можно решать гораздо более сложные задачи, чем с табличными компонентами.

В целом компонент запроса работает быстрее, т. к. если структура возвращаемых запросом полей может изменяться, то экземпляры класса `TFieldDef`, хранящие информацию о свойствах полей, создаются по необходимости при запуске приложения. Табличный компонент создает все классы для описания полей в любом случае, поэтому в приложениях "клиент-сервер" табличный компонент может открываться медленнее, чем запрос.

Рассмотрим общие свойства и методы компонентов запросов.

Текст запроса определяется свойством

```
property SQL: TStrings;
```

В свойстве

```
property Text: PChar;
```

содержится окончательно подготовленный текст запроса перед пересылкой его на сервер.

Выполнение запроса возможно тремя способами.

Если запрос возвращает результат в набор данных, то применяется метод

```
procedure Open;
```

или свойство

```
property Active: Boolean;
```

которому присваивается значение `True`. После выполнения запроса открывается набор данных компонента. Закрывается такой запрос методом

```
procedure Close;
```

или тем же свойством `Active`.

Если запрос не возвращает результат в набор данных (например, использует операторы `INSERT`, `DELETE`, `UPDATE`), то вызывается метод

```
procedure ExecSQL;
```

и после выполнения запроса набор данных компонента не открывается. Попытка использовать для такого запроса метод `Open` или свойство `Active` приведет к ошибке создания указателя на курсор данных.

После выполнения запроса в свойстве

```
property RowsAffected: Integer;
```

возвращается число обработанных при выполнении запроса записей.

Для того чтобы разрешить редактирование набора данных запроса, необходимо свойству

```
property RequestLive: Boolean;
```

присвоить значение `True`. Это свойство устанавливается, но не работает для запроса, результат которого не модифицируется из-за самого запроса.

Для подготовки запроса к выполнению предназначен метод

```
procedure Prepare;
```

который обеспечивает выделение необходимых ресурсов на сервере и проведение оптимизации.

Метод

```
procedure UnPrepare;
```

освобождает занятые при подготовке запроса ресурсы.

Результат выполнения этих двух операций отражается в свойстве

```
property Prepared: Boolean;
```

значение `True` говорит о том, что запрос подготовлен для выполнения.

Вызов методов `Prepare` и `UpPrepare` не является обязательным, т. к. компонент делает это автоматически. Однако если запрос будет выполняться несколько раз подряд, то подготовку необходимо провести перед первым выполнением запроса вручную. Тогда при последующих запросах сервер не будет тратить время на проведение бесполезной операции — ведь ресурсы под запрос уже были выделены.

Часто запросы имеют настраиваемые параметры, значения которых определяются непосредственно перед выполнением запроса.

Свойство

```
property Params: TParams;
```

представляет собой список объектов `TParam`, каждый из которых содержит настройки одного параметра. Свойство `Params` обновляется автоматически при изменении текста запроса. Подробнее о классе `TParams` рассказывается далее в этой главе.

Примечание

В компоненте `TADOQuery` свойство, аналогичное описанному свойству `Params`, называется `Parameters`.

Свойство

```
property ParamCount: Word;
```

возвращает число параметров запроса.

Свойство

```
property ParamCheck: Boolean;
```

определяет, необходимо ли обновлять свойство `Params` при изменении текста запроса во время выполнения. При значении `True` обновление осуществляется.

Кроме этого, компоненты запросов содержат еще некоторые свойства и методы, подробно описываемые в *главе 18*.

Компонент хранимой процедуры

Компонент хранимой процедуры предназначен для определения процедуры, установки ее параметров, выполнения процедуры и возвращения результатов в компонент.

В зависимости от выбранной технологии доступа к данным каждый компонент хранимой процедуры имеет собственный способ соединения с сервером. После подключения к источнику данных имя хранимой процедуры можно выбрать из списка свойства

```
property StoredProcName: String;
```

После этого свойство

```
property Params: TParams;
```

предназначенное для хранения параметров процедуры, автоматически заполняется.

Для хранимых процедур важно деление параметров на входные и выходные. Первые содержат исходные данные, а вторые передают результаты выполнения процедуры.

Детально класс `TParams` описывается далее.

Общее число параметров возвращает свойство

```
property ParamCount: Word;
```

Для подготовки хранимой процедуры служит метод

```
procedure Prepare;
```

или свойство

```
property Prepared: Boolean;
```

должно получить значение `True`.

Метод

```
procedure UnPrepare;
```

или свойство `Prepared := False` выполняют обратное действие.

Кроме того, проверка значения свойства `Prepared` позволяет установить, осуществлялась ли подготовка процедуры к выполнению или нет.

Внимание

После выполнения хранимой процедуры исходный порядок следования параметров в списке `Params` может измениться. Поэтому для доступа к конкретному параметру рекомендуется метод `function ParamByName(const Value: String): TParam;`

Если хранимая процедура возвращает набор данных, компонент можно открывать методом

```
procedure Open;
```

или свойством

```
property Active: Boolean;
```

В противном случае для выполнения процедуры необходим метод

```
procedure ExecProc;
```

и после этого выходные параметры получат вычисленные значения.

Индексы в наборе данных

Важнейшей проблемой для любой БД является достижение максимальной производительности и ее сохранение при дальнейшем увеличении объемов хранимых данных. Индексы позволяют решить эту задачу.

Индекс представляет собой часть базы данных, в которой содержится информация об организации данных в таблицах БД.

В отличие от ключей, которые просто идентифицируют отдельные записи, индексы занимают дополнительные объемы памяти (довольно значительные) и могут храниться как совместно с таблицами, так и в виде отдельных файлов. Индексы создаются вместе со своей таблицей и обновляются при модификации данных. При этом работа по обновлению индекса для большой таблицы может отнимать много ресурсов, поэтому имеет смысл ограничить число индексов для таких таблиц, где происходит частое обновление данных.

Индекс содержит в себе уникальные идентификаторы записей и дополнительную информацию об организации данных. Поэтому если при выполнении запроса сервер или локальная СУБД обращается для отбора записи к индексу, то это занимает значительно меньше времени, так как понятно, что идентификатор гораздо меньше самой записи. Кроме этого, индекс "знает", как организованы данные, и может ускорять обработку за счет группирования записей по сходным значениям параметров.

Создание для БД эффективного набора индексов является нетривиальной задачей.

Во-первых, нужно правильно определить оптимальное число индексов для каждой таблицы. Во-вторых, каждый индекс должен содержать только необходимые поля, при этом большую роль играет их упорядочивание.

В большинстве СУБД при создании индексов требуется только задать поля и название индекса, вся остальная работа выполняется автоматически.

Естественно, что в компонентах доступа к данным VCL Delphi реализованы все возможности такого мощного инструмента, как индексы. Причем свойства и методы для работы с индексами присутствуют только в табличных компонентах, т. к. в компонентах запросов работа с индексами осуществляется средствами SQL.

Набор данных может работать и без применения индексов, но для этого соответствующая таблица БД не должна иметь первичного ключа — случай довольно редкий. Поэтому по умолчанию в наборе данных установлен первичный индекс. При открытии набора данных все записи отсортированы в соответствии с первичным ключом.

Механизм подключения индексов

Для того чтобы подключить к набору данных вторичный индекс, необходимо присвоить свойству `IndexName` название индекса. Если свойство не имеет значения, то в наборе данных используется первичный индекс.

Альтернативный способ задания индекса заключается в применении свойства `IndexFieldNames`, в котором задается перечень имен полей необходимого индекса, разделенных точкой с запятой. При этом в Инспекторе объектов для этого свойства для существующих индексов список полей создается автоматически, разработчику остается сделать выбор. При помощи этого свойства можно создавать и составные индексы. Для этого необходимо, чтобы все входящие в список поля были индексированы.

Список имен всех индексов можно получить при помощи метода `GetIndexNames`.

Примечание

Изменение текущего индекса можно осуществлять без отключения набора данных, поэтому в приложениях очень удобно делать сортировку данных по индексам. Такой метод смены индексов называется индексацией "на лету".

После установки индекса количество полей в индексе передается в свойство `IndexFieldCount`.

Список описаний индексов

Информация об индексах набора данных содержится в свойстве класса `TDataSet`

```
property IndexDefs: TIndexDefs;
```

В нем для каждого индекса создается структура `TIndexDef`. Доступ к информации об индексах осуществляется через свойство

```
property Items[Index: Integer]: TIndexDef; default;
```

являющееся списком объектов TIndexDef.

Объекты типа TIndexDef можно добавлять в список при помощи метода

```
function AddIndexDef: TIndexDef;
```

Поиск объекта описания индекса осуществляет метод

```
function Find(const Name: String): TIndexDef;
```

который возвращает найденный объект по заданному в параметре Name имени индекса.

А пара методов

```
function FindIndexForFields(const Fields: string): TIndexDef;
function GetIndexForFields(const Fields: String; CaseInsensitive:
Boolean): TIndexDef;
```

находит объект описания индекса по списку полей, входящих в индекс. Если индекс не найден, ищется первый индекс, начинающийся с указанных полей. Первый из этих двух методов в случае неудачного поиска генерирует исключительную ситуацию EDatabaseError, а второй возвращает nil.

Список IndexDefs обновляется автоматически при открытии набора данных. Но метод

```
procedure Update; reintroduce;
```

обновляет список описаний индексов без открытия набора данных.

Описание индекса

Параметры каждого индекса набора данных представлены в классе TIndexDef, а их совокупность для набора данных содержится в свойстве IndexDefs класса TDataSet.

Свойство

```
property Name: String;
```

определяет название индекса.

Список всех полей индекса содержится в свойстве

```
property Fields: String;
```

Поля разделяются точкой с запятой.

Свойство

```
property CaseInsFields: String;
```

содержит список полей, регистр символов в которых при сортировке не учитывается. Поля разделяются точкой с запятой. Все поля из этого списка должны входить в свойство Fields. В наборе данных по умолчанию исполь-

зуется сортировка записей с учетом регистра символов. Но некоторые серверы БД допускают комбинированную сортировку по полям с учетом и без учета регистра.

Свойство

```
property DescFields: String;
```

содержит список через точку с запятой полей, которые сортируются в обратном порядке. Все поля из этого списка должны входить в свойство `Fields`. По умолчанию все поля сортируются в прямом порядке. Некоторые серверы БД поддерживают одновременную сортировку полей в прямом и обратном порядке.

Свойство

```
property GroupingLevel: Integer;
```

позволяет ограничить область применения индекса. Если значение этого свойства равно нулю, индекс упорядочивает все записи набора данных. В противном случае действие индекса распространяется на группы записей, имеющих одинаковые значения для того числа полей, которое задано свойством.

Параметры индекса определяются свойством

```
property Options: TIndexOptions;
```

Для индекса возможны сочетания следующих параметров:

- `ixPrimary` — первичный индекс;
- `ixUnique` — значения индекса уникальны;
- `ixDescending` — индекс сортирует записи в обратном порядке;
- `ixCaseInsensitive` — индекс сортирует записи без учета регистра символов;
- `ixExpression` — в индекс входит выражение (для индексов dBASE);
- `ixNonMaintained` — индекс не обновляется при открытии таблицы.

Метод

```
procedure Assign(ASource: TPersistent); override;
```

заполняет свойства объекта значениями аналогичных свойств объекта `ASource`.

Использование описаний индексов

Описания индексов наряду с описаниями полей *(см. главу 17)* также используются при создании новых таблиц БД. Для каждого планируемого индекса перед вызовом метода `CreateTable` необходимо создать или скопировать из

существующего набора данных соответствующее описание. Тогда при создании таблицы индексы будут добавлены автоматически (листинг 16.1).

Листинг 16.1. Пример использования описания индексов

```
with Table1 do
begin
 DatabaseName := 'DBDEMOS';
 TableType := ttParadox;
 TableName := 'DemoTable';
...
{Создание описаний полей}
...
 with IndexDefs do
 begin
  Clear;
  AddIndexDef;
  with Items[0] do
  begin
   Name := '';
   Fields := 'Field1';
   Options := [ixPrimary, ixUnique];
  end;
  AddIndexDef;
  with Items[1] do
  begin
   Name := 'SecondIndex';
   Fields := 'Field1;Field2';
   Options := [ixCaseInsensitive];
  end;
 end;
 CreateTable;
end;
```

При создании описаний индексов выбран метод `AddIndexDef`, который при каждом вызове добавляет к списку `Items` объекта `TIndexDefs` новый объект `TIndexDef`. Таким образом сначала создается первичный индекс (в таблицах Paradox он не имеет имени), затем вторичный `SecondIndex`. Для каждого описания обязательно определяются составляющие индекс поля и параметры индекса (свойства `Fields` и `Options`).

Параметры запросов и хранимых процедур

Свойство `Params` представляет собой набор изменяемых параметров запроса или хранимой процедуры. Свойство представляет собой набор объектов `TParam`, инкапсулирующих отдельные параметры.

Рассмотрим следующий запрос SQL:

```
SELECT SaleDat, OrderNo
FROM Orders
WHERE SaleDat >= '01.08.2001' AND SaleDat <= '31.08.2001'
```

В нем осуществляется отбор номеров заказов, сделанных в августе 2001 года. Теперь вполне естественно было бы предположить, что пользователю может понадобиться получить подобный отчет за другой месяц или за первые десять дней августа.

В этом случае можно поступить так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 with Query1 do
 begin
  SQL[0] := 'SELECT PartDat, ItemNo, ItemCount, InputPrice';
  SQL[1] := 'FROM Parts';
  SQL[2] := 'WHERE PartDat>= ''01.08.2005'' AND PartDat<=''31.08.2005''';
 end;
end;
procedure TForm1.RunBtnClick(Sender: TObject);
begin
 with Query1 do
 begin
  if Active then Close;
  SQL[2] := 'WHERE PartDat>= '+chr(39)+Date1Edit.Text+chr(39)+
  ' AND PartDat<='+chr(39)+Date2Edit.Text+chr(39);
  Open;
 end;
end;
```

При создании формы в методе `FormCreate` с помощью свойства `SQL` задается текст запроса. При щелчке на кнопке **`RunBtn`**, в соответствии с заданными в однострочных редакторах `Date1Edit` и `Date2Edit` датах, изменяется текст запроса. Метод `FormCreate` приведен только для того, чтобы обозначить первоначальный текст запроса, этот текст вполне можно задать в свойстве `SQL`.

Для решения подобных задач как раз и удобны параметры. В этом случае текст запроса будет выглядеть следующим образом:

```
SELECT PartDat, ItemNo, ItemCount, InputPrice
FROM Parts
WHERE PartDat>= :PD1 AND PartDat<= :PD2
```

Двоеточие перед именами `PD1` и `PD2` означает, что это параметры. Имя параметра выбирается произвольно. В списке свойства `Params` первым идет тот параметр, который расположен первым в тексте запроса.

После ввода в свойстве `SQL` текста запроса для каждого параметра автоматически создается объект `TParam`. Эти объекты доступны в специализированном редакторе, который вызывается при щелчке на кнопке свойства `Params` в Инспекторе объектов (рис. 16.2). Для каждого параметра требуется установить тип данных, который должен соответствовать типу данных соответствующего поля.

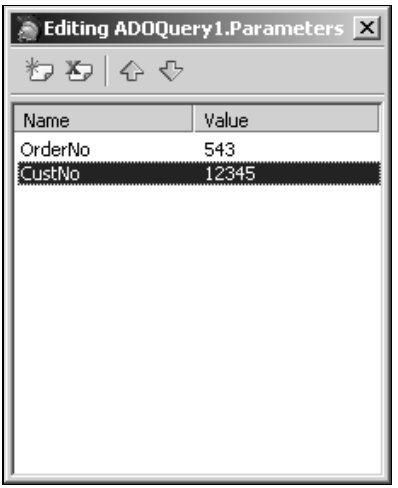


Рис. 16.2. Специализированный редактор параметров запроса

Теперь для задания текущих ограничений по дате поступления можно применить свойство `Params`:

```
procedure TForm1.RunBtnClick(Sender: TObject);
begin
 with Query1 do
 begin
  Close;
```

```
  Params[0].AsDateTime := StrToDate(Date1Edit.Text);
  Params[1].AsDateTime := StrToDate(Date2Edit.Text);
  Open;
 end;
end;
```

При щелчке на кнопке **RunBtn** при помощи параметров в запрос передаются текущие значения ограничений дат.

Значения параметров запроса можно задать и из другого набора данных. Для этого имеется свойство `DataSource` компонента набора данных. Указанный в свойстве компонент `TDataSource` должен быть связан с набором данных, значения полей которого требуется передать в параметры. Названия параметров должны соответствовать названиям полей этого набора данных, тогда свойство `DataSource` начнет работать. При перемещении по записям набора данных текущие значения одноименных параметрам полей автоматически передаются в запрос.

Для иллюстрации работы этого свойства рассмотрим простой пример, главная форма которого представлена на рис. 16.3. Этот проект не имеет ни одной строки написанного вручную программного кода.

Верхний компонент `TDBGrid` отображает данные из таблицы `Orders` базы данных DBDEMOS и связан через компонент `OrdersSource` типа `TDataSource` с компонентом запроса `OrdersQuery`. Текст запроса выглядит так:

```
SELECT * FROM Orders
```

Нижний компонент `TDBGrid` отображает данные о покупателе и через компонент `CustSource` типа `TDataSource` связан с запросом `CustSource`, свойство `SQL` которого имеет следующий вид:

```
SELECT * FROM Customer
WHERE CustNo=:CustNo
```

Обратите внимание, что название параметра соответствует названию поля номера покупателя из таблицы `Orders`.

Свойство `DataSource` компонента `CustQuery` указывает на компонент `OrdersSource`.

Как только оба набора данных открываются, текущее значение из поля `CustNo` набора данных `Orders` автоматически передается в параметр запроса компонента `CustQuery`.

Благодаря этому для двух наборов данных реализована связь "один к одному" по полю номера поставщика.

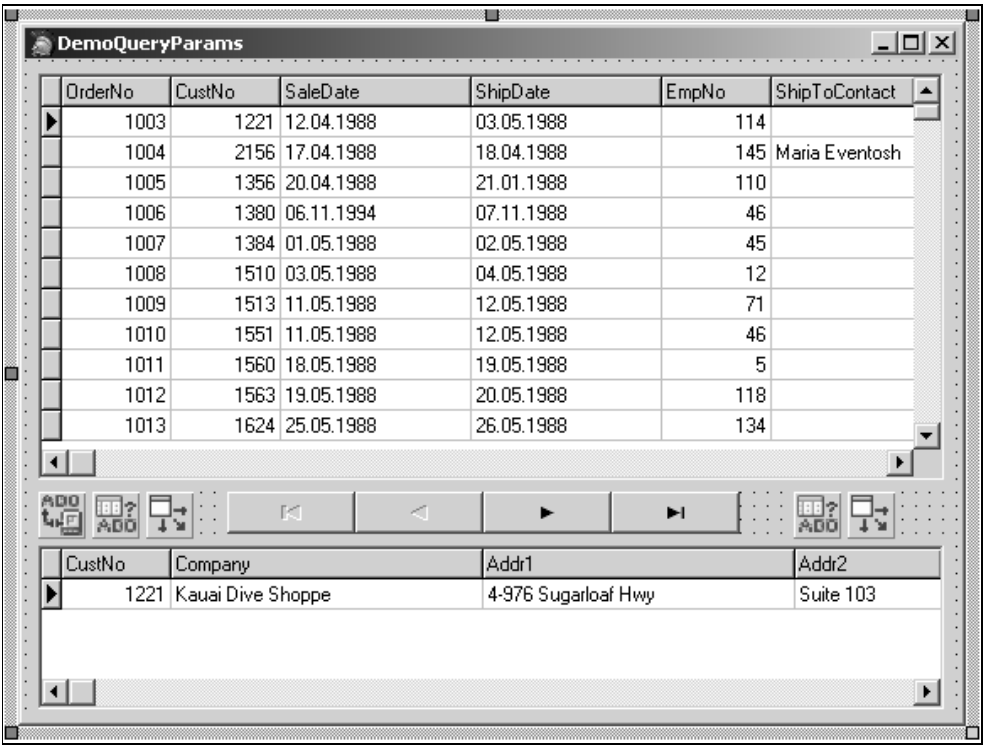


Рис. 16.3. Главная форма проекта DemoQueryParams

И в завершение разговора о параметрах запросов рассмотрим свойства и методы класса `TParams`, составляющего свойство `Params`, и класса `TParam`, инкапсулирующего отдельный параметр.

Класс *TParams*

Класс `TParams` представляет собой список параметров.

Доступ к элементам списка возможен через индексированное свойство

```
property Items[Index: Word]: TParam;
```

а к значениям параметров через свойство

```
property ParamValues[const ParamName: String]: Variant;
```

Добавить новый параметр можно методом

```
procedure AddParam(Value: TParam);
```

Но для него необходимо создать объект параметра. Это можно сделать методом

```
function CreateParam(FldType: TFieldType; const ParamName: string;
ParamType: TParamType): TParam;
```

Здесь

- `FldType` — тип данных;
- `ParamName` — имя;
- `ParamType` — тип параметра.

И оба метода можно использовать в связке:

```
MyParams.AddParam(MyParams.CreateParam(ftInteger, 'Param1', ptInput));
```

Вместо того, чтобы заполнять параметры по одному, можно вызвать метод

```
function ParseSQL(SQL: String; DoCreate: Boolean): String;
```

который при `DoCreate = True` анализирует текст запроса из `SQL` и создает новый список параметров.

Для присвоения значений сразу всем параметрам удобен метод

```
procedure AssignValues(Value: TParams);
```

Для удаления параметра из списка применяется метод

```
procedure RemoveParam(Value: TParam);
```

При работе с параметрами для их идентификации целесообразнее обращение по имени, т. к. при работе с хранимыми процедурами после их выполнения порядок следования может измениться. Это справедливо и для динамических запросов (их текст SQL может изменяться во время выполнения).

Для обращения к параметру по имени существует метод

```
function ParamByName(const Value: String): TParam;
```

В сложных запросах SQL или после многочисленных исправлений разработчик может допустить ошибку и создать два разных параметра с одним именем. В этом случае при выполнении запроса одноименные параметры считаются одним, и им присваивается значение первого по порядку запроса. Для контроля повторных имен в списке параметра предусмотрен метод

```
function IsEqual(Value: TParams): Boolean;
```

который возвращает `True`, если для параметра `Value` найден дубликат.

Класс *TParam*

Класс `TParam` инкапсулирует свойства отдельного параметра.

Имя параметра определяется свойством

```
property Name: String;
```

Тип данных параметра задает свойство

```
property DataType: TFieldType;
```

Тип данных параметра и связанного поля должны совпадать.

Тип параметра определяется множеством

```
type
  TParamType = (ptUnknown, ptInput, ptOutput, ptInputOutput, ptResult);
  TParamTypes = set of TParamType;
```

которое имеет следующие значения:

- `ptUnknown` — тип неизвестен;
- `ptInput` — параметр предназначен для передачи значения из приложения;
- `ptOutput` — параметр предназначен для передачи значения в приложение;
- `ptInputOutput` — параметр предназначен для передачи и приема значения;
- `ptResult` — параметр предназначен для передачи в приложения информации о статусе операции.

А свойство

```
property ParamType: TParamType;
```

определяет тип параметра.

При работе с параметрами довольно часто бывает необходимо определить, имеет ли параметр ненулевое значение. Для этого есть свойство

```
property IsNull: Boolean;
```

Свойство возвращает `True`, если параметр не имеет значения или имеет значение `Null`.

А свойство

```
property Bound: Boolean;
```

возвращает `True` только тогда, когда параметру не присваивалось значение вообще.

Метод

```
procedure Clear;
```

присваивает параметру значение `Null`.

Само значение параметра задается свойством

```
property Value: Variant;
```

Но использование вариантов не очень эффективно, когда требуется обеспечить максимальную скорость. В таких случаях можно обратиться к целому

набору свойств As..., которые не только возвращают значение, но и приводят его к некоторому типу. Например, свойство

```
property AsInteger: LongInt;
```

возвращает целочисленное значение поля.

Примечание

Необходимо осторожно использовать свойства с приведением типа, т. к. попытка преобразования неверного значения вызовет исключительную ситуацию.

Для чтения из буфера и записи в буфер значения параметра соответственно имеются методы

```
procedure SetData(Buffer: Pointer);
procedure GetData(Buffer: Pointer);
```

а необходимый размер при записи в буфер позволит определить метод

```
function GetDataSize: Integer;
```

Можно скопировать тип данных, имя и значение параметра прямо из поля данных. Для этого применяется метод

```
procedure AssignField(Field: TField);
```

а для присвоения типа данных и значения служит метод

```
procedure AssignFieldValue(Field: TField; const Value: Variant);
```

Общее число знаков для числовых значений определяет свойство

```
property Precision: Integer;
```

А свойство

```
property NumericScale: Integer;
```

задает число знаков после запятой.

Для строковых параметров размер задает свойство

```
property Size: Integer;
```

Состояния набора данных

В процессе своего функционирования от открытия методом Open и до закрытия методом Close набор данных может выполнять самые разнообразные операции. Можно просто перемещаться по записям, можно редактировать данные и удалять записи, можно проводить поиск по различным параметрам и т. д. При этом желательно, чтобы все операции выполнялись как можно быстрее и эффективнее.

Набор данных в любой момент времени находится в некотором состоянии, т. е. подготовлен к выполнению действий строго определенного рода. И для каждой группы операций набор данных выполняет ряд подготовительных действий.

Все состояния набора данных делятся на две группы.

- Состояния, в которые набор данных переходит автоматически; к ним относятся непродолжительные по времени состояния, сопровождающие функционирование полей набора данных (табл. 16.1).
- Состояния, которыми можно управлять из приложения, например, перевод набора данных в режим редактирования (табл. 16.2).

Базовый класс `TDataSet`, инкапсулирующий свойства набора данных, позволяет изменять состояние, а также проверять текущее состояние набора данных.

Текущее состояние набора данных передается в свойство `State`, имеющее тип `TDataSetState`:

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead,
dsInternalCalc);
```

Для управления состояниями набора данных существуют методы `Open`, `Close`, `Edit` и `Insert`.

Таблица 16.1. *Автоматические состояния набора данных*

Константа состояния	Описание
`dsNewValue`	Включается при обращении к свойству `NewValue` поля набора данных
`dsOldValue`	Включается при обращении к свойству `OldValue` поля набора данных
`dsCurValue`	Включается при обращении к свойству `CurValue` поля набора данных
`dsInternalCalc`	Включается при расчете значений полей, для которых `FindKind = fkInternalCalc`
`dsCalcFields`	Включается при выполнении метода `OnCalcFields`
`dsBlockRead`	Включается механизм ускоренного перемещения по набору данных
`dsOpening`	Существует при открытии набора данных методом `Open` или свойством `Active`
`dsFilter`	Включается при выполнении метода `OnFilterRecord`

Таблица 16.2. *Управляемые состояния набора данных*

Константа состояния	Метод	Описание
`dsInactive`	`Close`	Набор данных закрыт
`dsBrowse`	`Open`	Данные доступны для просмотра, но недоступны для редактирования
`dsEdit`	`Edit`	Данные можно редактировать
`dsInsert`	`Insert`	К набору данных можно добавлять новые записи
`dsSetKey`	`SetKey`	Включается механизм поиска по ключу. Также могут использоваться диапазоны

Рассмотрим, как изменяется состояние набора данных при выполнении стандартных операций.

Закрытый набор данных всегда имеет неактивное состояние `dsInactive`.

При открытии набор данных переходит в состояние просмотра данных `dsBrowse`. В этом состоянии по записям набора данных можно перемещаться и просматривать их содержимое, но редактировать данные нельзя. Это основное состояние открытого набора данных, из него можно перейти в другие состояния, но любое изменение состояния происходит через просмотр данных (рис. 16.4).

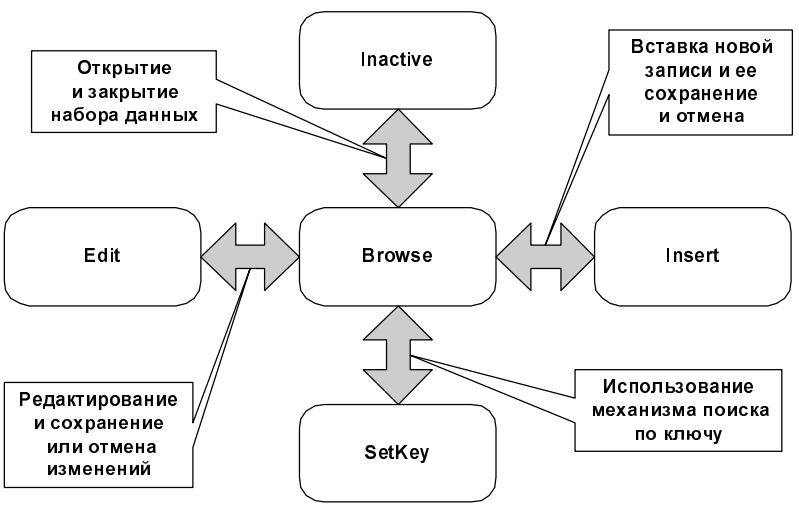


Рис. 16.4. Схема изменения состояний набора данных

При необходимости редактирования данных набор должен быть переведен в состояние редактирования `dsEdit`, для этого предназначен метод `Edit`. После выполнения метода можно изменять значения полей для текущей записи. При перемещении на следующую запись набор данных автоматически переходит в состояние просмотра.

Для того чтобы вставить в набор данных новую запись, необходимо установить состояние вставки `dsInsert`. Метод `Insert` переводит набор данных в это состояние и добавляет на месте текущего курсора новую пустую запись. При переходе на другую запись, после проверки на уникальность первичного ключа (если он есть), набор данных возвращается в состояние просмотра.

Состояние установки ключа `dsSetKey` имеется только в табличных компонентах при необходимости поиска методами `FindKey` и `FindNext`, а также при использовании диапазонов (метод `SetRange`). Это состояние сохраняется до момента вызова одного из методов поиска по ключу или метода отмены диапазона. После этого набор данных возвращается в состояние просмотра.

Состояние просмотра по блокам `dsBlockRead` целесообразно для реализации быстрого перемещения по большим массивам записей без показа промежуточных записей в компонентах отображения данных и без вызова обработчика события перемещения по записям. Для быстрого перемещения по набору данных можно использовать методы `DisableControls` и `EnableControls`.

Резюме

Набор данных является образом таблицы базы данных в приложении. Он содержит группу записей и обеспечивает их применение.

Класс `TDataSet`, инкапсулирующий функциональность набора данных, является базовым классом для всех технологий доступа к данным. На его основе созданы все основные компоненты, применяемые при разработке приложений баз данных. Условно их можно разделить на три группы:

- компоненты таблиц;
- компоненты запросов;
- компоненты хранимых процедур.

В этой главе рассмотрены важнейшие свойства, методы и структуры, реализованные в компонентах, инкапсулирующих набор данных.

Глава 17

Поля и типы данных

Каждая таблица БД и, следовательно, набор данных приложения имеет собственную структуру, которая определяется совокупностью полей. Любое *поле* набора данных представляет собой объект, содержащий описание типа данных, которому должно соответствовать значение, находящееся в записи на определенном месте. Иначе, полем можно назвать совокупность ячеек с данными определенного типа, расположенные в одном и том же месте каждой записи набора данных или, попросту, это столбец в таблице.

В наборе данных приложения баз данных Delphi каждому полю соответствует собственный объект. Основой объектов полей является класс `TField`, который инкапсулирует основные свойства абстрактного поля, не зависящего от типа данных. От этого базового класса порождены другие класса, обеспечивающие функционирование реальных объектов полей, зависящих от типа данных.

Программист, грамотно использующий возможности полей, может решать существенно более сложные задачи и создавать эффективные и гибкие приложения баз данных.

Эта глава посвящена изучению объектов полей набора данных и приемов работы с ними. В главе рассматриваются следующие вопросы.

- Объект поля в наборе данных.
- Динамические и статические объекты полей.
- Способы применения объектов полей в наборе данных.
- Класс `TField` — основа работы с полями в наборах данных.
- Типы объектов полей и типы данных.
- Ограничения.

Объекты полей

Объекты полей инкапсулируют свойства и методы полей различных типов данных. Они функционируют совместно с набором данных и очень тесно

связаны с ним. Например, для того, чтобы получить значения полей из текущей записи набора данных, разработчик должен создать примерно такой код:

```
Edit1.Text := Table1.Fields[0].AsString;
```

Свойство `Fields` представляет собой индексированный список объектов полей набора данных *(см. главу 16)*. Если разработчик не изменяет порядок следования полей в наборе данных, то расположение объектов полей в списке `Fields` соответствует структуре таблицы базы данных.

Каждый объект полей хранит ряд параметров, определяющих поле. Например, в наборе данных к объекту поля можно обратиться, зная только название поля:

```
Edit1.Text := Table1.FieldByName('SomeField').AsString;
```

Для того чтобы присвоить значение полю в текущей записи, можно воспользоваться приведенными способами, или, если тип данных поля неизвестен, свойством `FieldValues`:

```
Table1.FieldValues['SomeField'] := Edit1.Text;
```

Знание имени поля дает самый простой способ обращения к его текущему значению:

```
Table1['SomeField'] := Edit1.Text;
Edit1.Text := Table1['SomeField'];
```

Примечание

При присваивании значений полям набора данных необходимо контролировать состояние, в котором находится набор данных *(см. главу 16)*.

В основе классов, описывающих иерархию типизированных полей, лежит класс `TField`. От него порождены другие классы, обеспечивающие работу целых групп полей, объединенных по типам данных (рис. 17.1).

Что же такое объект поля и какие возможности он предоставляет разработчику?

Во-первых, назначение класса `TField`, как базового класса поля, заключается в умении взаимодействовать с компонентом отображения данных для обеспечения правильной визуализации данных. Например, объект поля хранит способ выравнивания, параметры шрифта, текст заголовка и т. д.

Во-вторых, с точки зрения набора данных, объект поля является хранилищем текущего значения этого поля (а не всего столбца данных, как это можно себе представить по названию).

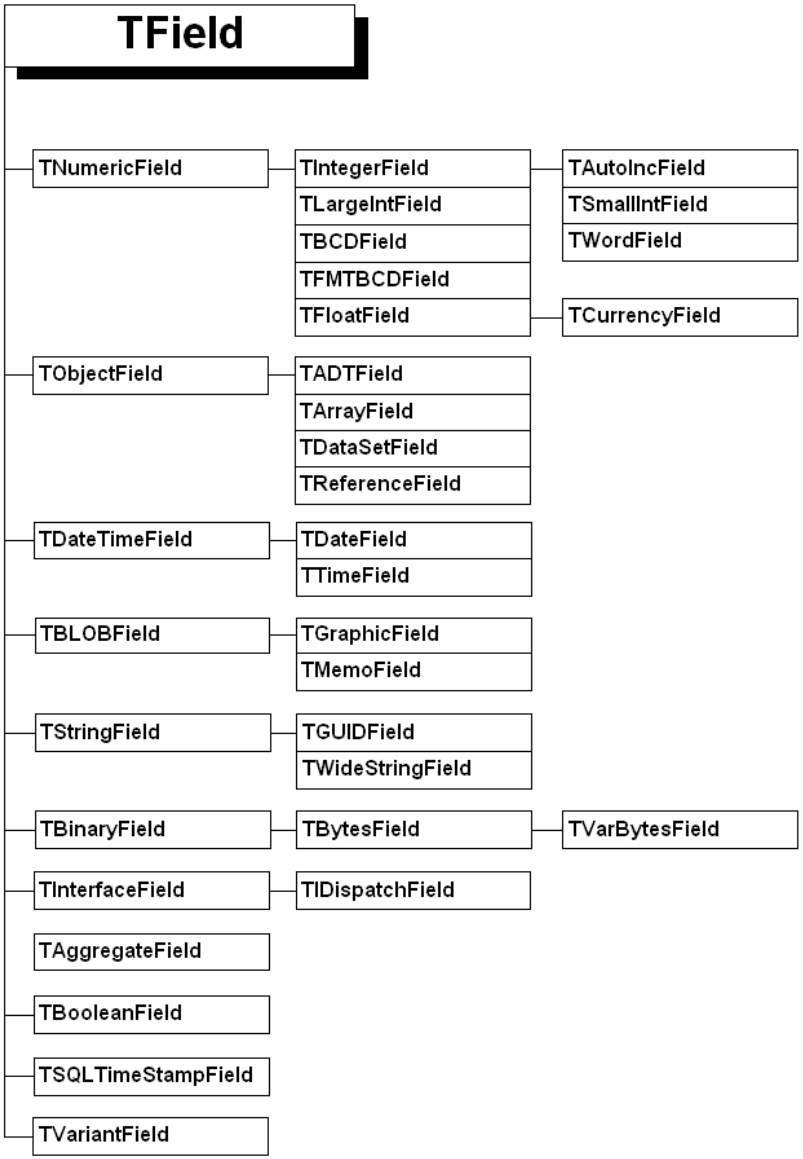


Рис. 17.1. Иерархия классов полей

Компоненты отображения данных при работе с набором данных взаимодействуют именно с полями. Например, колонки компонента `TDBGrid` при отсутствии дополнительных настроек соответствуют расположению объектов полей в связанном наборе данных.

Статические и динамические поля

В Delphi предусмотрено два способа создания объектов полей.

Динамические поля используются программой в случае, если разработчик не создал для них объекты явным образом на этапе разработки. Каждый не заданный явно объект поля автоматически создается при открытии набора данных в соответствии со структурой связанной таблицы БД. Любой объект поля является прямым наследником класса `TField`, а его конкретный тип зависит от типа данных поля таблицы. При этом свойства динамического поля устанавливаются в соответствии с параметрами поля таблицы базы данных.

Компонент набора данных после подключения к таблице БД без дополнительных настроек задействует только динамические поля. К свойствам и методам динамических полей можно обратиться программно, для этого предусмотрены индексированное свойство `Fields` компонента доступа к данным, которое объединяет все поля набора данных или метод `FieldByName`.

Динамические поля удобны в случаях, когда заданные характеристики полей в таблице базы данных полностью удовлетворяют разработчика и нет необходимости рассматривать какое-либо поле вне набора данных.

Статические поля создаются программистом на этапе разработки, их свойства доступны в Инспекторе объектов, а их названия можно выбрать из списка объектов активной формы в верхней части Инспектора объектов. Название статического объекта поля обычно складывается из названий таблицы и поля, например, `OrdersCUSTNO`.

Создаются статические объекты полей при помощи специализированного Редактора полей, который вызывается двойным щелчком на компоненте набора данных на форме или командой **Fields Editor** из всплывающего меню этого компонента (рис. 17.2).

Редактор полей представляет собой простой список уже созданных статических полей, все управление осуществляется командами из всплывающего меню. В верхней части окна Редактора расположены кнопки навигатора для перемещения по набору данных, которые активны только при открытом наборе данных. Если набор данных имеет агрегатные поля данных, то они размещаются в отдельном списке в нижней части окна Редактора полей.

Добавить к списку статических новое поле, существующее в таблице БД, можно при помощи команды **Add fields** из всплывающего меню Редактора. Удаление элемента из списка осуществляется клавишей <Delete>. Перетаскиванием элементов списка при помощи мыши можно изменить их расположение. Таким образом, можно создавать произвольные комбинации статических полей.

Рис. 17.2. Редактор полей
с отдельным списком агрегатных полей

Примечание

Как только для набора данных создан хотя бы один статический объект поля, считается, что набор данных содержит только те поля, которые имеются в списке статических полей. Эту особенность можно использовать для искусственного ограничения структуры данных таблиц. Все поля набора данных расположены в том порядке, как это указано в списке Редактора полей, независимо от их реального положения в таблице базы данных.

Команда **New field** из всплывающего меню Редактора полей позволяет создать статическое поле, которое реально не существует в структуре данных таблицы (рис. 17.3). Для выбора типа поля предусмотрена группа радиокнопок **Field Type**: **Data** — поле данных, **Calculated** — вычисляемое поле, **Lookup** — поле синхронного просмотра.

Реже создаются поля данных, которые обязательно должны базироваться на реальных полях таблицы. Если для такого объекта соответствующее поле в таблице будет удалено или его тип данных будет изменен, то при открытии набора данных генерируется исключительная ситуация.

Примечание

Для клиентских наборов данных многоуровневых приложений диалог создания нового поля позволяет выбрать два дополнительных типа поля: агрегатные поля (радиокнопка **Aggregate**) и внутренние вычисляемые поля (радиокнопка **InternalCalc**).

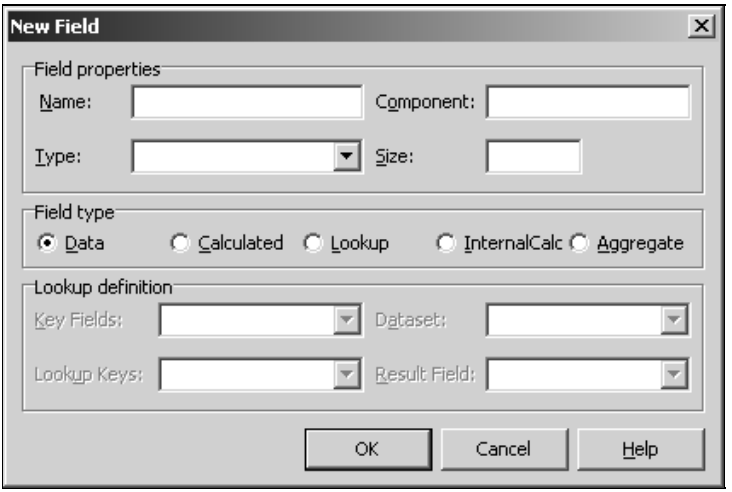


Рис. 17.3. Диалог создания нового статического поля Редактора полей набора данных

Класс *TField*

Как уже говорилось, в большой иерархии классов для полей различных типов данных класс `TField` является базовым (см. рис. 17.1), он инкапсулирует свойства и методы абстрактного поля данных. Именно от него происходят все классы типизированных полей. В реальной работе класс `TField` не используется, но его значение трудно переоценить. Практически все основные свойства классов типизированных полей унаследованы от класса `TField` без каких-либо изменений, а дополнительные свойства и методы обеспечивают работу конкретного типа данных.

Что касается методов-обработчиков событий, то четыре метода, определенные в классе `Tfield`, наследуются всеми потомками без изменения и дополнения.

Далее приведены свойства и методы класса `TField`.

Имя объекта содержит свойство

```
property Name: TComponentName;
```

При создании объекта поля на этапе разработки имя объекта складывается из имени соответствующего компонента набора данных и имени поля.

Свойство

```
property FieldName: String;
```

возвращает имя поля таблицы базы данных.

Свойство

```
property FullName: string;
```

используется, если текущее поле является дочерним для другого поля. В таком случае свойство содержит имена всех родительских полей.

Название поля в таблице базы данных содержится в свойстве

```
property Origin: String;
```

Свойство

```
property FieldNo: Integer;
```

возвращает исходный порядковый номер поля в наборе данных. Если объекты полей являются статическими, их фактический порядок может быть изменен в Редакторе полей. А свойство

```
property Index: Integer;
```

содержит индекс объекта поля в списке `Fields`.

Функциональное назначение поля определяется свойством

```
type TFieldKind = (fkData, fkCalculated, fkLookup, fkInternalCalc,
fkAggregate);
property FieldKind: TFieldKind;
```

В большинстве случаев его значение определяется автоматически в момент создания объекта поля. Да и впоследствии вряд ли возникнет необходимость сделать реальное поле данных вычисляемым. Обычно попытка изменить значение свойства `FieldKind` вызывает ошибку. Рассмотрим возможные значения этого свойства:

- `fkData` — поле данных;
- `fkCalculated` — вычисляемое поле;
- `fkLookup` — поле синхронного просмотра;
- `fkInternalCalc` — внутреннее вычисляемое поле;
- `fkAggregate` — агрегатное поле.

Если поле является вычисляемым, свойство

```
property Calculated: Boolean;
```

принимает значение `True`.

На связанный набор данных указывает свойство

```
property DataSet: TDataSet;
```

которое при создании объекта средствами среды разработки заполняется автоматически.

Свойство

```
property DataType: TFieldType;
```

возвращает тип данных поля, а свойство

```
property DataSize: Integer;
```

содержит объем памяти, необходимый для хранения значения поля.

Одной из важнейших задач класса `TField` является обеспечение доступа к текущему значению поля. В этом случае класс взаимодействует с буфером текущей записи набора данных, а значение можно получить при помощи нескольких свойств.

Свойство

```
property Value: Variant
```

всегда содержит значение, которое сохранено после последнего выполнения метода `Post` набора данных (листинг 17.1).

Листинг 17.1. Использование свойства `Value`

```
with Table1 do
begin
 Open;
 while Not EOF do
 begin
  if Fields[0].Value > 10
  then Fields[1].Value := Fields[1].Value*2;
  Next;
 end;
 Close;
end;
```

В этом примере при помощи метода `Next` осуществляется перебор всех записей набора данных. Если значение первого поля больше 10, то значение второго поля удваивается. Для этого предусмотрено свойство `Value` объектов полей набора данных.

Однако из-за использования вариантов свойство `Value` является относительно медленным. Для преобразования текущего значения поля к необходимому виду имеется целая группа быстрых свойств `As...`, которые содержат значение в определенном типе данных. Чаще всего употребляется свойство `AsString`, например, оно может применяться для представления числовых значений полей в элементах управления:

```
Edit1.Text := Table1.Fields[0].AsString;
```

Примечание

При работе со статическими объектами полей при передаче значений целесообразнее свойства из группы `As...`, т. к. неявное задание типа свойством `Value` может привести к ошибке преобразования данных типа вариант.

Если свойство

```
property CanModify: Boolean;
```

имеет значение `False`, значения поля нельзя редактировать. Однако это свойство является только средством для определения возможности редактирования. А свойство

```
property ReadOnly: Boolean;
```

позволяет запретить редактирование (`ReadOnly := True`) или разрешить его (`ReadOnly := False`).

Большая группа свойств отвечает за представление и форматирование значения поля.

Свойство

```
property DisplayText: String;
```

содержит значение поля в строковом формате до начала редактирования.

Свойство

```
property Text: String;
```

предназначено для использования компонентами отображения данных при редактировании. Поэтому эти два свойства могут иметь разные значения в случае, если значение поля в строковом формате при редактировании и просмотре различно. У классов-наследников `TField` для этого достаточно задать шаблон отображения данных для поля (свойство `DisplayFormat`) и шаблон редактирования данных (свойство `EditFormat`). Например, вещественное число при просмотре может иметь разделители тысяч, а при редактировании нет. В этом случае рассматриваемые свойства будут иметь следующий вид:

```
DisplayText = '1 452,32'
Text = '1452,32'
```

Свойства `Text` и `DisplayText` влияют на использование метода-обработчика `OnGetText`. Если параметр `DisplayText` имеет значение `True`, то параметр `Text` содержит значение свойства `DisplayText`, в противном случае в метод передается значение поля в строковом формате.

Если поле не имеет значения, то при помощи свойства `DefaultExpression` можно задать некоторое постоянное значение, которое будет появляться в компоненте отображения данных при пустом поле. Если постоянное значе-

ние содержит какие-либо символы кроме цифр, то все выражение нужно обязательно брать в кавычки.

В случае возникновения исключительных ситуаций во время работы с полем генерируется соответствующее сообщение, в котором в качестве имени поля используется значение свойства `DisplayName`. Если задано свойство `DisplayLabel`, то `DisplayName` приравнивается к нему, в противном случае для задания свойства `DisplayName` используется свойство `FieldName`. Другим способом задать значение свойства `DisplayName` невозможно.

Свойство

```
property DisplayWidth: Integer;
```

определяет число символов для отображения значений поля в визуальных компонентах отображения данных.

Свойство

```
property Visible: Boolean;
```

отвечает за видимость поля в визуальных компонентах отображения данных. При этом компоненты, отображающие одно поле, перестают показывать его значения, а компоненты типа `TDBGrid` не отображают колонки, связанные с полем.

Примечание

Еще несколько групп свойств класса `TField`, а также его методы-обработчики рассматриваются ниже в этой главе.

Виды полей

Теперь рассмотрим классификацию полей набора данных в зависимости от их функционального назначения. Самыми распространенными являются поля данных, базирующиеся на реальных полях таблицы БД. Свойства объектов таких полей устанавливаются в соответствии с параметрами полей таблицы БД.

Кроме этого, в практике программирования часто применяются поля синхронного просмотра и вычисляемые поля.

Процесс создания всех типов полей набора данных практически не отличается. Тем не менее такое разнообразие позволяет успешно решать самые сложные задачи программирования приложений БД.

Далее мы рассмотрим только поля синхронного просмотра и вычисляемые поля, т. к. поля данных не содержат каких-либо существенных особенностей в применении.

С точки зрения набора данных, большой разницы между этими двумя видами полей нет. Однако значения для всех полей синхронного просмотра рассчитываются раньше, чем для вычисляемых полей. Поэтому вы можете использовать поля синхронного просмотра в выражениях вычисляемых полей и не можете сделать наоборот.

Поля синхронного просмотра

При создании для исходного набора данных нового поля синхронного просмотра необходимы следующие свойства.

Свойство

```
property LookupDataSet: TDataSet;
```

задает набор данных синхронного просмотра.

Свойство

```
property LookupResultField: String;
```

представляет поле синхронного просмотра из набора данных `LookupDataSet`, данные из которого будут появляться в созданном поле.

Свойство

```
property LookupKeyFields: String;
```

содержит поле (или поля) из набора данных синхронного просмотра, по значению которого выбирается значение из поля `LookupResultField`.

Свойство

```
property KeyFields: String;
```

определяет поле (или поля) из исходного набора данных, для которого создается поле синхронного просмотра.

Для визуализации данных из поля синхронного просмотра допустимы любые компоненты отображения данных, но естественно будет применить специальные компоненты синхронного просмотра, о которых рассказывается в *главе 19*.

Кроме этого, очень удобно использовать поля синхронного просмотра в компоненте `TDBGrid`. Если такое поле связать с одной из колонок компонента, то для него автоматически заполняется список синхронного просмотра. Его элементы хранятся в свойстве `PickList`, которое имеется в любой колонке. Теперь пользователю достаточно выбрать нужную колонку в сетке и, щелкнув на появившейся в текущей ячейке кнопке, получить возможные значения для замены. Одновременно с изменением поля синхронного просмотра изменяется и ключевое поле (свойство `KeyFields`) исходного набора данных.

Примечание

При использовании полей синхронного просмотра в компоненте `TDBGrid` открывать набор данных синхронного просмотра необязательно. При этом свойство `LookupCache`, о котором далее пойдет речь, обязательно должно иметь значение `False`.

Идентифицировать поля синхронного просмотра можно с помощью булевского свойства `Lookup` базового класса `TField`, которое принимает истинное значение для таких полей.

Свойство

```
property LookupCache: Boolean;
```

определяет режим функционирования специального буфера значений синхронного просмотра. Если это свойство истинно, то буфер работает.

Буфер основан на свойстве `LookupList`. При открытии исходного набора данных каждое поле синхронного просмотра получает свое значение, одновременно с этим в соответствии со всеми имеющимися в исходном наборе данных значениями ключевого поля заполняется и буфер синхронного просмотра. Впоследствии, при перемещении на другую запись значение берется не из набора данных синхронного просмотра, а из буфера. Этот механизм при небольших объемах значений синхронного просмотра позволяет увеличить скорость работы с исходным набором данных, особенно в режиме удаленного доступа при низкоскоростных сетях.

При изменениях в наборе данных синхронного просмотра можно применить метод

```
procedure RefreshLookupList;
```

который обновляет текущее значение поля и список значений в буфере.

Специально для разработчиков в базовый класс `TField` включено свойство

```
property Offset: Integer;
```

которое возвращает размер буфера в байтах.

Для создания поля синхронного просмотра удобнее всего воспользоваться Редактором полей компонента доступа к данным. После выбора команды **New field** из всплывающего меню в одноименном диалоге (см. рис. 17.3), помимо обычных действий, соответствующих созданию поля данных, необходимо задать значения свойств в группе **Lookup definition**. Элементы управления группы становятся доступными после выбора типа поля (радиокнопка **Lookup** в группе **Field type**).

В списке **Dataset** представлены все доступные в модуле наборы данных, из которых нужно выбрать набор данных синхронного просмотра (свойство `Lookup DataSet`). Список **Result Field** позволяет выбрать поле синхронного просмотра (свойство `LookupResultField`). Список **Lookup Keys** задает клю-

чевое поле в наборе данных синхронного просмотра (свойство LookupKeyFields). Список **Key Fields** определяет ключевое поле исходного набора данных (свойство KeyFields).

Вычисляемые поля

Вычисляемые поля существенно облегчают разработку приложений баз данных, т. к. позволяют получать новые данные на основе существующих, не изменяя при этом структуру таблиц БД. Выражения для получения значений вычисляемых полей разработчик должен разместить в методе-обработчике OnCalcFields набора данных. Здесь допустимы любые арифметические и логические операции и функции, любые операторы языка, свойства и методы любых компонентов, в том числе запросы SQL (листинг 17.2).

Листинг 17.2. Пример работы с вычисляемыми полями

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
 with Table1 do
 Table1CalcField1.Value := Fields[0].Value + Fields[1].Value;
 with Query1 do
 begin
  Params[0].AsInteger := Table1.Fields[0].AsInteger;
  Open;
  Table1CalcField1.Value := Fields[0].AsString;
  Close;
 end;
end;
```

Метод OnCalcFields выполняется при открытии набора данных, при переходе в режим редактирования, при передаче фокуса между компонентами отображения данных или колонок сетки, при удалении записи. Но для этого нужно, чтобы свойство AutoCalcFields набора данных было равно True.

Примечание

Необходимо учитывать, что сложные вычисляемые поля могут существенно замедлить работу набора данных (особенно при наличии запросов SQL). Кроме того, в процессе редактирования набора данных (при изменении значения поля, сохранении изменений и переходе на следующую запись) вычисляемые поля рассчитываются несколько раз подряд. Для уменьшения числа автоматических обращений к методу OnCalcFields нужно использовать свойство AutoCalcFields := False.

Для создания вычисляемого поля достаточно в диалоге создания нового поля Редактора полей в качестве типа поля задать вычисляемое, в остальном процесс совпадает с созданием поля данных.

В выражениях вычисляемых полей могут присутсвовать другие вычисляемые поля, но они обязательно должны быть определены в методе `OnCalcFields` до этого.

Вычисляемые поля недопустимы при фильтрации набора данных методом-обработчиком `OnFilterRecord`, т. к. он вызывается до метода-обработчика `OnCalcFields`, а вычисляемые поля не сохраняются.

Внутренние вычисляемые поля

Помимо простых, существуют внутренние вычисляемые поля (`FieldKind = fkInternalCalc`). Они встречаются в клиентских наборах данных (компоненты `TClientDataSet`) и отличаются тем, что их значения сохраняются в наборе данных.

Внутренние вычисляемые поля пригодны для фильтрации методом-обработчиком `OnFilterRecord`.

Агрегатные поля

Агрегатные поля предназначены для выполнения вычислительных операций со значениями полей набора данных с использованием агрегатных функций SQL. К таким функциям относятся:

- `AVG` — вычисляет среднее значение;
- `COUNT` — возвращает число записей;
- `MIN` — вычисляет минимальное значение;
- `MAX` — вычисляет максимальное значение;
- `SUM` — вычисляет сумму.

Агрегатные поля не входят в структуру полей набора данных, т. к. агрегатные функции подразумевают объединение записей таблицы для получения результата. Следовательно, значение агрегатного поля нельзя связать с какой-то одной записью, оно относится ко всем или группе записей.

Агрегатные поля допустимы только в компоненте `TClientDataSet` и его аналогах, т. к. он обеспечивает кэширование данных, необходимое для проведения вычислений *(см. главу 29)*.

Агрегатные поля не отображаются вместе со всеми полями в компонентах `TDBGrid`, в Редакторе полей расположены в отдельном списке, а их свойство `Index` всегда имеет значение −1. Для представления значения агрегатного поля можно воспользоваться одним из компонентов отображения данных,

который визуализирует значение одного поля (например, TDBText или TDBEdit) или свойствами самого поля:

```
Label1.Caption := MyDataSetAGGRFIELD1.AsString;
```

Создать агрегатное поле можно командой **New field** из всплывающего меню Редактора полей.

Для представления агрегатных полей имеется специальный класс TAggregateField.

Его свойство

```
property Expression: string;
```

задает вычисляемое выражение. В его состав могу входить агрегатные функции, имена полей набора данных и простейшие арифметические операции:

```
SUM(Pirce*ItemCount) — SUM(Balance)
```

Вычисление значения проводится только для тех агрегатных полей, свойство

```
property Active: Boolean;
```

которых имеет значение True.

Вычисление включенных свойством Active агрегатных полей выполняется только в том случае, если булевское свойство AggregatesActive клиентского компонента набора данных имеет значение True.

По умолчанию экземпляр класса TAggregateField создается со свойством Visible = False.

Свойство

```
property GroupingLevel: Integer;
```

задает уровень группировки полей набора данных при вычислении. При значении 0 расчет проводится для всех записей набора данных. При значении 1 записи группируются по первому полю набора данных и расчет осуществляется для каждой группы. При значении 2 записи разбиваются на группы по первому и второму полям и т. д.

Однако группировка по уровням выше нулевого возможна, только если в наборе данных есть индекс по группирующим полям. Например, если свойство GroupingLevel = 2 и набор данных начинается с полей CustNo и OrderNo, в свойстве IndexName компонента набора данных и свойстве

```
property IndexName: String;
```

объекта агрегатного поля должно быть имя индекса, включающего оба эти поля.

Из-за необходимости подключения индексов уровень группировки выше нулевого возможен только в табличных компонентах.

Объектные поля

Наряду с обычными типами данных (строковым, целочисленным и т. д.), при работе с полями набора данных возможны более сложные, представляющие собой совокупность простых.

В Delphi существуют четыре класса объектных полей. Это `TADTField`, `TArrayField`, `TDataSetField` и `TReferenceField`. Их общим предком является класс `TObjectField`. Классы `TADTField`, `TArrayField` обеспечивают доступ к набору дочерних полей одного типа из родительского. Эти типы полей можно использовать, если сервер БД поддерживает объектные типы и соответствующие поля имеются в наборе данных. Поэтому объектные поля можно создавать статически и динамически, так же, как и простые поля.

Для доступа к дочерним полям в этих классах имеются свойства

```
property Fields: TFields;
```

которое представляет собой индексированный список объектов дочерних полей *(см. главу 16)*, и

```
property FieldValues[Index: Integer]: Variant;
```

содержащее значения дочерних полей.

Свойство

```
property FieldCount: Integer;
```

возвращает число дочерних полей.

Классы `TDataSetField` и `TReferenceField` предоставляют доступ к данным из связанных наборов данных.

Ссылка на набор данных задается свойством

```
property DataSet: TDataSet;
```

Более подробно классы `TDataSetField` и `TReferenceField` рассматриваются в *части V*.

Типы данных

В среде разработки Delphi можно создавать приложения для работы с самыми разными базами данных. Такая универсальность подразумевает необходимость применения средств, которые бы обеспечили возможность работы со многими типами данных этих баз данных.

Естественно, что существует большая группа типов данных, конкретная реализация которых практически не отличается от платформы к платформе. Это, например, строки, символы, целые и вещественные числа и т. д.

Есть типы данных, которые реализованы далеко не на каждой платформе. Существуют, наконец, просто уникальные типы данных.

Для удовлетворения потребностей разработчиков в Delphi применен следующий способ работы с типами данных.

Тип данных однозначно связан с конкретным полем таблицы базы данных. Без этого поля само понятие типа данных не имеет практического смысла. В Delphi свойства абстрактного поля инкапсулирует класс `TField`, который не имеет заранее определенного типа данных. Уже от этого класса порождено целое семейство классов для типизированных полей (см. рис. 17.1), каждый из которых умеет обращаться со своим типом данных.

Примечание

В классе `TField` имеется свойство `DataType`, которое отвечает за тип данных, но оно не может быть изменено.

Весь список доступных типов данных содержится в типе `TFileldType`:

```
type TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime,
ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar,
ftWideString, ftLargeint, ftADT, ftArray, ftReference, ftDataSet,
ftOraBlob, ftOraClob, ftVariant, ftInterface, ftIDispatch, ftGuid,
ftTimeStamp, ftFMTBcd);
```

В табл. 17.1 приведены все типы данных, которые можно использовать при разработке приложений для работы с базами данных.

Таблица 17.1. *Типы данных*

Тип	Класс	Описание
Неизвестный (`FtUnknown`)		Неопределенный тип данных
Строковый (`ftString`)	`TStringField`	Строка длиной до 8192 символов
Целый короткий (`ftSmallint`)	`TSmallIntField`	16-битное целое в диапазоне от –32 768 до 32 767
Целый (`ftInteger`)	`TIntegerField`	32-битное целое в диапазоне от –2 147 483 648 до 2 147 483 647
Целый положительный (`ftWord`)	`TWordField`	16-битное целое в диапазоне от 0 до 65 535

Таблица 17.1 *(продолжение)*

Тип	Класс	Описание
Логический (ftBoolean)	TBooleanField	Значения True и False
Вещественный (ftFloat)	TFloatField	Вещественные положительные и отрицательные числа с точностью 15 цифр после запятой в диапазоне от $5{,}0\times10^{-324}$ до $1{,}7\times10^{308}$
Денежный (ftCurrency)	TCurrencyField	Вещественные положительные и отрицательные числа с точностью 15 цифр после запятой в диапазоне от $5{,}0\times10^{-324}$ до $1{,}7\times10^{308}$. Дополнительно вставляется символ валюты
Десятичный с двоичным кодированием (ftBCD)	TBCDField	Вещественные числа с повышенной точностью (до 4 знаков перед запятой и до 20 знаков после запятой). Могут храниться в двоичном и десятичном форматах
Дата (ftDate)	TDateField	Дата
Время (ftTime)	TDateTimeField	Время
Календарный (ftDateTime)	TDateTimeField	Комбинированный формат с одновременным хранением даты и времени
Фиксированный буфер (ftBytes)	TBytesField	Набор байтов фиксированного размера. Для работы с этим типом требуется выделять и освобождать память (методы GetMem и FreeMem)
Переменный буфер (ftVarBytes)	TVarBytesField	Набор байтов переменного размера. Текущий размер буфера хранится в первых двух байтах. Для работы с этим типом требуется выделять и освобождать память (методы GetMem и FreeMem)
Автоинкрементный (ftAutoInc)	TAutoIncField	Значение поля в каждой новой записи автоматически увеличивается на 1. Целое число в диапазоне от –2 147 483 648 до 2 147 483 647. Применяется для обеспечения уникальности значений ключей
BLOB (ftBlob)	TBLOBField	Большой двоичный массив. Используется для хранения любых данных, которые можно преобразовать в цифровой массив (Memo, Graphic). В базах данных такие данные хранятся в отдельных файлах, а поле содержит лишь ссылки на них

Таблица 17.1 *(продолжение)*

Тип	Класс	Описание
Memo (`ftMemo`)	`TMemoField`	Набор строк произвольной длины
Графический (`ftGraphic`)	`TGraphicField`	Формат для хранения изображений
Форматирован-ный Memo (`ftFmtMemo`)		Форматированный набор строк произ-вольной длины
OLE Paradox (`ftParadoxOle`)		Поле OLE для таблиц Paradox
OLE dBASE (`ftDBaseOle`)		Поле OLE для таблиц dBASE
Типизирован-ный двоичный (`ftTypedBinary`)		Типизированный двоичный
Курсор Oracle (`ftCursor`)		Курсор для хранимых процедур сервера Oracle
Фиксирован-ный символьный (`ftFixedChar`)	`TStringField`	Строка символов с нулевым символом в конце
Расширенный строковый (`ftWideString`)		Динамически выделяемая строка 16-битных символов в кодировке Unicode
Целый большой (`ftLargeint`)	`TLargeIntField`	64-битное целое число
Абстрактный (`ftADT`)	`TADTField`	Произвольный тип данных, создаваемый пользователем на сервере БД и исполь-зуемый в приложении
Массив (`ftArray`)	`TArrayField`	Массив полей любого типа, кроме `TArrayField`
Ссылочный (`ftReference`)	`TReferenceField`	Указатель на объект, содержащийся в другой таблице
Набор данных (`ftDataSet`)	`TDataSetField`	Содержит набор данных, интегрированный в текущий набор данных
BLOB Oracle 8 (`ftOraBlob`)		Тип CLOB для сервера Oracle 8

Таблица 17.1 *(окончание)*

Тип	Класс	Описание
CLOB Oracle 8 (`ftOraClob`)		Тип CLOB для сервера Oracle 8
Вариант (`ftVariant`)	`TVariantField`	Вариант
Интерфейс (`ftInterface`)	`TInterfaceField`	Ссылка на интерфейс (потомок от `IUnknown`)
Ссылка на интерфейс IDispatch (`ftIDispatch`)	`TIDispatchField`	Сссылка на интерфейс (потомок от `IDispatch`)
Глобальный идентификатор (`ftGuid`)	`TGuidField`	Глобальный идентификатор GUID
Календарный (`ftTimeStamp`)		Календарный тип для наборов данных dbExpress
Десятичный с двоичным кодированием (`ftFMTBcd`)	`TFMTBCDField`	Тип BCD повышенной точности

Как видно из таблицы, наряду с традиционными типами данных, в Delphi имеются и специальные типы, благодаря которым значительно расширяется функциональность приложений. В частности, типы `ftInterface`, `ftIDispatch`, `ftGuid` позволяют создавать полноценные приложения БД для COM и OLE DB.

Практически во всех серверах БД пользователь имеет возможность создавать собственные типы данных. Для их использования в приложениях Delphi имеется абстрактный тип данных и класс `TADTField`. Абстрактный тип может включать любой скалярный тип данных (числа, даты, ссылки, массивы и наборы данных).

Автоинкрементный тип данных давно применяется в СУБД. Поле автоинкрементного типа для каждой новой записи автоматически увеличивает свое значение на единицу. Тем самым, каждая запись имеет собственный уникальный идентификатор, который очень часто является первичным ключом.

Данные типа BLOB (Binary Large OBject) представляют собой двоичные массивы произвольной длины. В самом поле содержится лишь ссылка на отдельный файл базы данных, в котором хранится двоичный массив. Таким образом, поля типа BLOB являются универсальным носителем любых дан-

ных, которые имеют скалярную и нескалярную структуру и которые можно преобразовать в двоичное представление.

Тип `Memo` представляет собой набор строк произвольной длины (его разновидность — форматированный `Memo`), основан на формате BLOB. Удобен при необходимости сохранить текст из компонента `TMemo` или из текстового редактора.

Графический тип данных пригоден для хранения в базе данных изображений, он основан на формате BLOB. Поле `TGraphicField` непосредственно взаимодействует с компонентом отображения данных (например, `TDBImage`). Изображения должны храниться в формате BMP.

Типы данных `ParadoxOle` и `dBaseOle` разработаны специально для СУБД Paradox и dBASE по работе с данными OLE. В Delphi эти типы данных основаны на формате BLOB.

Специально для работы с сервером Oracle 8 предназначены типы CLOB и BLOB.

Тип `ftArray` организует массив из данных любой структуры, за исключением таких же массивов. Для каждого элемента массива может создаваться собственный объект `TField`. Для управления этим механизмом предусмотрено свойство `SparseArrays` в классе `TDataSet`.

В качестве отдельного поля в набор данных можно включить и любой другой произвольный набор данных. Для этого существует специальный тип данных и класс `TDataSetField`. Причем каждым полем из интегрированного набора данных также можно управлять.

Ссылочный тип данных также задействует внешние наборы данных, но в данном случае можно подключать и использовать только отдельные поля.

Ограничения

Контроль вводимых в поля набора данных в Delphi возложен на объекты полей, а не на компоненты отображения данных. Именно в рамках этого вопроса мы рассмотрим работу методов-обработчиков событий базового класса `TField`.

Перед сохранением значения поля в БД всегда вызывается метод-обработчик `OnValidate`, при этом автоматически проводится проверка на выполнение задаваемых ограничений и ограничений типа данных. Кроме этого, здесь можно предусмотреть свою дополнительную обработку:

```
procedure TForm1.Table1SomeFieldValidate(Sender: TField);
begin
 if (Sender as TField).Value < 0 then
```

```
 begin
  ShowMessage('Значение поля не может быть отрицательным');
  (Sender as TField).Value := Null;
 end;
end;
```

Различают контроль значения в целом и посимвольный контроль. Метод `OnValidate` проверяет значение поля целиком. Если при проверке обнаружена ошибка, то выдается сообщение и фокус формы устанавливается на соответствующем компоненте отображения данных.

Если метод `OnValidate` не вызвал исключительной ситуации, то при сохранении значения поля в БД вызывается обработчик события `OnChange`. В принципе, можно предусмотреть операции по контролю данных и в этом методе, но тогда в случае ошибки возникает нежелательная исключительная ситуация, которая может привести к серьезным сбоям в работе приложения.

Проверить текущее значение поля перед его появлением в компоненте отображения данных можно в методе-обработчике `OnGetText`. Если параметр `DisplayText` принимает истинное значение, то в параметре `Text` передается значение свойства `DisplayText` (значение в строковом формате в таком виде, как оно будет показано в компоненте отображения данных — с символами форматирования). В противном случае в параметре `Text` передается текущее значение в строковом формате.

Примечание

В случае метода-обработчика `OnGetText` на разработчика ложится обязанность самостоятельно предусмотреть передачу значения в компонент отображения данных, иначе компонент останется незаполненным.

В методе-обработчике `OnSetText` можно осуществлять текущий контроль значения в строковом формате в том виде, как оно представлено в компоненте отображения данных. Напомним, что этот обработчик вызывается при каждом изменении свойства `Text` класса `TField`.

Рассмотренные методы-обработчики удобнее всего для проверки текущего значения с точки зрения программной логики. Например, чтобы отпускная цена не была ниже закупочной или чтобы остаток не был больше первоначального количества товара в партии. Для проверки правильности самого значения класс `TField` имеет несколько полезных свойств.

Если на сервере БД задано ограничение на некоторое поле, его можно использовать в приложении Delphi при помощи свойства `ImportedConstraint`

Создать собственное ограничение можно с помощью свойства `CustomConstraint` с синтаксисом SQL:

```
Value>10
```

или

```
OutputPrice>InputPrice*1.25
```

При возникновении ошибки совсем не лишним будет, если программа выдаст некое осмысленное сообщение, которое поможет пользователю исправить оплошность. При работе с методами-обработчиками это можно предусмотреть в программном коде.

Для встроенного контроля предусмотрено специальное свойство — `ConstraintErrorMessage`, которое выводится в виде сообщения при возникновении ошибки. Согласитесь, что это гораздо проще, чем исправлять и перекомпилировать соответствующие файлы ресурсов. Если приложение работает с сервером БД и возникла ошибка ограничения поля, то выводится сообщение, определяемое сервером, а не этим свойством.

Если для поля заданы ограничения, то свойство `HasConstraints` принимает истинное значение.

Посимвольный контроль данных осуществляется свойством `ValidChars`, в котором можно определить допустимые в строковом представлении значения поля символы, и методом `IsValidChar`, который определяет допустимость переданного в параметре символа.

Еще один мощный инструмент контроля данных предоставляет свойство `EditMask`, которое позволяет создавать шаблоны ввода данных, облегчая тем самым работу пользователя и уменьшая возможность ошибки. Рассмотрим правила создания шаблонов.

Шаблон состоит из трех частей.

Первая часть содержит управляющие символы собственно шаблона. Доступные для создания шаблона символы приведены в табл. 17.2.

Таблица 17.2. *Управляющие символы шаблона*

Символ	Описание
!	
>	Все символы после этого преобразуются в заглавные
<	Все символы после этого преобразуются в строчные
<>	Все символы после этого остаются в том регистре, как это было задано пользователем
\	Символ, следующий за этим, считается алфавитным, а не управляющим
L	В позиции этого символа обязательно должен находиться только алфавитный символ

Таблица 17.2 *(окончание)*

Символ	Описание
I	В позиции этого символа может находиться алфавитный символ
A	В позиции этого символа обязательно должен находиться алфавитный символ или цифра
a	В позиции этого символа может находиться алфавитный символ или цифра
C	В позиции этого символа обязательно должен находиться знак препинания
c	В позиции этого символа может находиться знак препинания
0	В позиции этого символа обязательно должна находиться цифра
9	В позиции этого символа может находиться цифра
#	В позиции этого символа может находиться цифра, плюс или минус
:	Символ разделения часов, минут и секунд (зависит от системных установок)
/	Символ разделения дней, месяцев, годов (зависит от системных установок)
;	Символ разделения частей шаблона
_	Символ автоматического ввода в текст пробела

В первую часть шаблона можно включать любые алфавитные символы (для создания поясняющих надписей, слов и сокращений), если их нет среди управляющих символов. В качестве алфавитных допустимы также и управляющие символы, для этого перед ними нужно поместить символ \.

Вторая часть состоит из одного символа и определяет, могут ли неарифметические символы быть частью вводимого текста. Если здесь расположен ноль, то можно вводить только цифры, если любой другой символ — то и алфавитные символы.

В третьей части содержится символ, служащий для обозначения мест, запрещенных для ввода.

Части шаблона разделяются точкой с запятой.

Например, шаблон для ввода телефонного номера выглядит следующим образом:

```
!\(999\)000-0000;1;_
```

Резюме

Работа с полями является важным этапом в процессе разработки приложения баз данных. Для этого предусмотрены специальные объекты, которые инкапсулируют возможности полей таблицы БД. В Delphi имеется целая иерархия классов, обеспечивающая применение полей самых различных типов. В основе этой иерархии лежит класс `TField`.

По способу создания объекты полей делятся на статические и динамические.

По функциональным возможностям объекты полей бывают полями данных, вычисляемыми, синхронного просмотра и агрегатными.

Объекты полей играют важную роль в работе наборов данных. С их помощью можно получить доступ к текущим значениям, задать ограничения на вводимые величины и проверить их правильность.

Глава 18

Управление данными

Наряду с описанными в предыдущих главах свойствами и методами, стандартный набор данных Delphi инкапсулирует ряд дополнительных механизмов, облегчающих управление записями и полями.

К ним относятся такие полезные функции, как быстрое перемещение по записям, поиск нужной записи по значениям полей, дополнительная фильтрация записей набора данных без использования возможностей СУБД и т. д. Большинство этих механизмов работают с индексами таблиц БД.

Абстрактные методы, обеспечивающие управление данными, реализованы в базовом классе `TDataSet` *(см. главу 16)*. А классы-потомки, в свою очередь, реализуют механизмы управления данными в соответствии с возможностями технологий доступа к данным *(см. часть 5)*.

Все рассматриваемые в этой главе методы управления данными в полном объеме доступны только в компонентах, инкапсулирующих таблицу БД. Это связано с тем, что компоненты запросов SQL и хранимых процедур не обеспечивают полноценное использование индексов.

В этой главе рассматриваются следующие вопросы.

- Связанные таблицы.
- Методы поиска данных.
- Диапазоны.
- Быстрая навигация по набору данных.
- Фильтрация записей в наборе данных.

Связанные таблицы

В рамках одного проекта таблицы БД можно связывать отношениями "один ко многим" и "многие ко многим", при этом отношения обязательно устанавливаются между индексированными полями двух таблиц.

При создании отношений в качестве главной таблицы можно выбрать любой компонент, инкапсулирующий набор данных. Для задания подчиненной таблицы пригодны только табличные компоненты *(см. главу 16)*.

Отношение "один ко многим"

Для установления отношения "один ко многим" в наборе данных предназначены два свойства `MasterSource` и `MasterFields`, которые задаются для подчиненной таблицы. Набор данных главной таблицы не требует никаких дополнительных настроек и заданная связь будет работать только при перемещениях по записям главной таблицы.

Свойство `MasterSource` определяет компонент `TDataSource`, который связан с главной таблицей.

Затем при помощи свойства `MasterFields` необходимо установить отношения между полями главной и подчиненной таблицы. В нем содержится имя индексированного поля, по которому устанавливается связь. Если таких полей несколько, их имена разделяются точкой с запятой. При этом не все поля, входящие в индекс, обязаны участвовать в создании отношения.

Для задания свойства `MasterFields` можно использовать Редактор связей полей (Field Link Designer), который вызывается щелчком на кнопке в поле редактирования этого свойства в Инспекторе объектов (рис. 18.1).

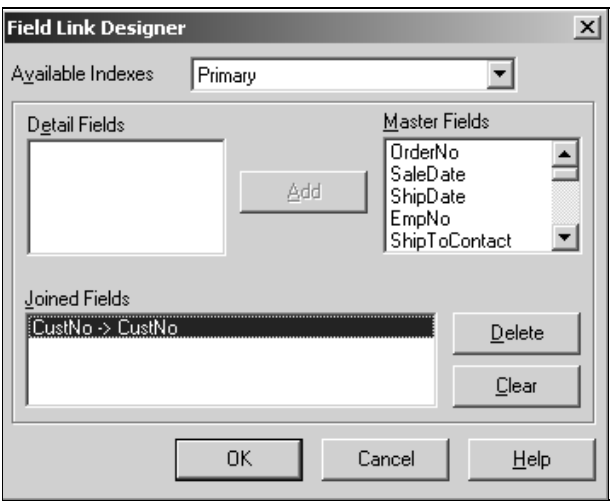


Рис. 18.1. Редактор связей полей

Здесь в раскрывающемся списке **Available Indexes** выбирается требуемый индекс для подчиненной таблицы. После этого в поле **Detail Fields** появляются

имена всех полей, входящих в этот индекс. В списке **Master Fields** отображаются все поля главной таблицы.

Теперь требуется создать связи между полями. Для этого в левом списке выбирается поле подчиненной таблицы, а затем соответствующее ему поле главной таблицы в правом списке. После этого активизируется кнопка **Add**, щелчок на которой создает отношение по двум полям главной и подчиненной таблиц. Созданная связь отображается в списке **Joined Fields**.

Примечание

После создания связи по индексированным полям данный индекс становится текущим для набора данных. При этом в зависимости от типа СУБД автоматически заполняются свойства `IndexName` или `IndexFieldNames`.

Уже созданные связи можно удалить. Кнопка **Delete** удаляет выбранную связь, кнопка **Clear** удаляет все связи.

После создания связей между полями отношение "один ко многим" считается установленным. Теперь достаточно открыть оба набора данных, чтобы увидеть работу отношения.

В качестве примера рассмотрим проект DemoJoins (рис. 18.2), в котором связываются таблицы из демонстрационной базы данных DBDEMOS. Для этого использованы компоненты ADO, подробнее о которых вы можете узнать из *главы 23*.

Таблица `Customers` представлена в наборе данных компонента `CustTable`, она содержит данные о покупателях. Таблица `Orders` представлена в наборе данных компонента `OrdTable`, она содержит данные о заказах. Таблица `Employee` представлена в наборе данных компонента `EmpTable`, она содержат данные о продавцах.

Примечание

Приложение DemoJoins не содержит дополнительного исходного кода. Все отношения между таблицами заданы при помощи Инспектора объектов.

Отношение "один ко многим" задано между таблицами покупателей (`Customers`) и заказов (`Orders`). Таблица покупателей является главной. Для создания отношения установлены следующие значения свойств компонента `OrdTable` (подчиненная таблица).

Свойство `MasterSource` должно указывать на компонент `CustSource`, связанный с набором данных `CustTable`.

Свойство `MasterFields` указывает на поле `CustNo` таблицы `Customers`.

В наборе данных `OrdTable` включен вторичный индекс на основе поля `CustNo` (`IndexName = 'CustNo'`).

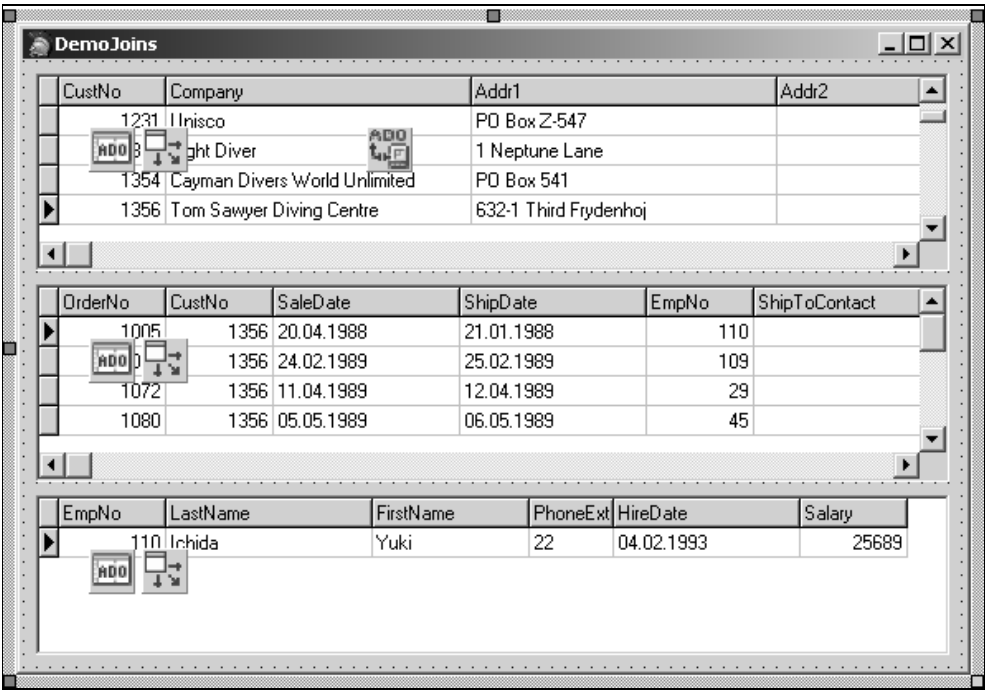


Рис. 18.2. Главная форма проекта DemoJoins

Таким образом, две таблицы связаны отношением "один ко многим" по индексированным полям `CustNo` (номер покупателя). В результате, при перемещении по записям таблицы покупателей в таблице заказов будут показаны только те заказы, которые относятся к текущему покупателю.

Отношение "многие ко многим"

Отношение "многие ко многим" отличается тем, что подчиненная таблица еще раз связывается в качестве главной с другой подчиненной таблицей аналогичной последовательностью действий, как и в отношении "один ко многим".

В приложении DemoJoins отношением "многие ко многим" связаны таблицы заказов (`Orders`) и продавцов (`Employee`). Таблица заказов уже работает в отношении "один ко многим" в качестве подчиненной.

В наборе данных `EmpTable` заданы следующие свойства.

Свойство `MasterSource` указывает на компонент `EmpSource`.

Свойство MasterFields содержит имя поля EmpNo, по которому осуществляется связь между таблицами. Для подчиненной таблицы поле EmpNo является первичным.

Поиск данных

В наборе данных реализованы два способа поиска записей по заданным значениям полей. Первый основан на механизме индексов и является более быстрым, но поиск проводится только по индексированным полям. Второй способ использует специальные методы классов наборов данных и позволяет проводить поиск по любому сочетанию полей, но он более медленный.

Поиск по индексам

Для организации индексного поиска к набору данных должен быть подключен индекс (свойства IndexName или IndexFieldNames).

Метод FindKey проводит поиск записи по заданным в параметре значениям ключевых полей текущего индекса набора данных. В случае успеха курсор набора данных устанавливается на найденной записи, а метод возвращает значение True. В противном случае возвращается False.

Если индекс состоит из нескольких полей, значения для поиска записываются в виде множества, причем отсутствующие значения приравниваются к Null.

Рассмотрим простейший пример (рис. 18.3, листинг 18.1), в котором реализован поиск по вторичному индексу в таблице CUSTOLY.DB демонстрационной базы данных DBDEMOS. Индекс основан на полях Last_Name и First_Name.

В компоненте Table1, помимо стандартных настроек на таблицу, при помощи свойства IndexName задан и вторичный индекс (его имя Names). Значения для поиска задаются в компонентах Edit1 и Edit2.

Листинг 18.1. Секция implementation главного модуля Main проекта DemoFind

```
implementation
{$R *.DFM}
procedure TForm1.FormShow(Sender: TObject);
begin
 try
  Cust.Open;
 except
```

```
  on E: EDBEngineError do ShowMessage('Ошибка при открытии таблицы');
 end;
end;
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 Cust.Close;
end;
procedure TForm1.FindBtnClick(Sender: TObject);
begin
 try
 if not Cust.FindKey([Edit1.Text, Edit2.Text])
  then ShowMessage('Запись не найдена');
 except
  on E: EDatabaseError
   do ShowMessage('Ошибка поиска');
 end;
end;
end.
```

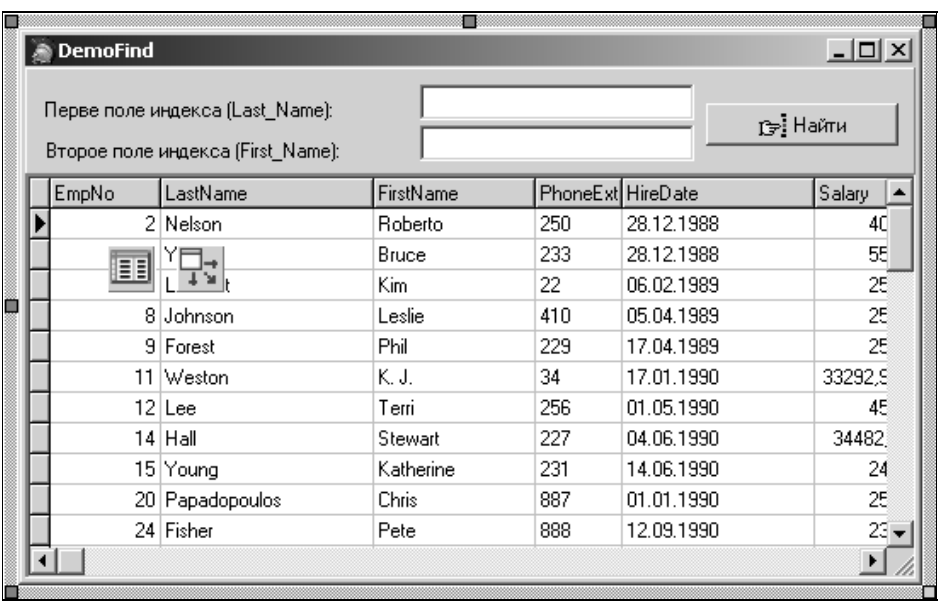


Рис. 18.3. Главная форма проекта DemoFind

Набор данных открывается в методе-обработчике FormShow при открытии формы и закрывается в методе-обработчике FormClose. При щелчке на кнопке FindBtn в метод FindKey передаются значения для поиска из компонентов Edit1 и Edit2.

Поиск в диапазоне

Кроме этого, индексный поиск можно организовать группой методов, подобно созданию диапазонов. Метод SetKey переводит набор данных в состояние dsSetKey, затем должно следовать присваивание ключевым полям значений для поиска. Сам поиск осуществляется методом GotoKey:

```
with Table1 do
begin
 SetKey;
 Fields[0].Value := '428';
 GotoKey;
end;
```

В случае успеха курсор набора данных устанавливается на найденной записи, а метод возвращает значение True. Вместо этого метода можно применять метод GotoNearest, который в случае неудачного поиска ищет запись, минимально отличающуюся от критерия поиска.

Изменение параметров поиска осуществляется методом EditKey.

Поиск по произвольным полям

Для поиска по произвольной выборке полей имеются методы Locate и Lookup.

```
function Locate(const KeyFields: string; const KeyValues: Variant;
Options: TLocateOptions): Boolean;
```

```
function Lookup(const KeyFields: string; const KeyValues: Variant; const
ResultFields: string): Variant;
```

В метод Locate необходимо передать список полей, по которым будет идти поиск (параметр KeyFields, имена полей разделяются точкой с запятой), их требуемые значения (параметр KeyValues, значения разделяются запятой) и настройки поиска (параметр Options). В настройках можно задать опцию loCaseInsensitive, которая отключает проверку на регистр символов и опцию loPartialKey, которая включает поиск с минимальными отличиями. В случае успеха поиска курсор набора данных устанавливается на найденной записи, а метод возвращает значение True.

```
Table1.Locate('Last_Name;First_Name', VarArrayOf(['Edit1.Text',
'Edit2.Text']), []);
```

В метод `Lookup` передается список полей для поиска (параметр `KeyFields`, имена полей разделяются точкой с запятой) и их требуемые значения (параметр `KeyValues`, значения разделяются запятой). В случае успешного поиска функция возвращает массив значений типа вариант для полей, названия которых содержатся в параметре `ResultFields`.

```
Table1.Lookup('Last_Name;First_Name', VarArrayOf(['Edit1.Text',
'Edit2.Text']), 'Last_Name;First_Name');
```

Оба эти метода автоматически используют быстрый индексный поиск в случае, если в параметре `KeyFields` задать поля индекса.

Фильтры

Наиболее эффективным способом отбора записей в набор данных (особенно из больших таблиц) является создание и выполнение соответствующего запроса SQL. Но что делать, если набор данных функционирует на базе табличного компонента? В этом случае на помощь приходит встроенный в набор данных механизм фильтрации данных.

Применение фильтра основано всего на двух основных свойствах и одном вспомогательном. Текст фильтра должен содержаться в свойстве `Filter`, а свойство `Filtered` включает и выключает фильтр. Параметры фильтра определяются свойством `FilterOptions`.

Примечание

Компонент `TQuery` также может использовать фильтры. Эта возможность подчас позволяет легко и изящно решать довольно сложные проблемы, которые иначе требуют изменения текста запроса или создания нового компонента запроса.

При вызове фильтра его текст транслируется в синтаксис SQL и передается для выполнения на сервер или через соответствующий драйвер в локальную СУБД.

Фильтры можно создавать двумя способами.

- При помощи свойства `Filter` создаются довольно простые фильтры, для которых достаточно предоставляемого механизмом фильтрации синтаксиса.
- Для создания более сложных фильтров с применением всех возможных средств языка программирования применяется метод-обработчик набора данных `OnFilterRecord`.

Фильтры можно разделить на статические и динамические.

Статические фильтры создаются во время разработки приложения и могут использовать как свойство `Filter`, так и метод `OnFilterRecord`.

Динамические фильтры можно создавать и редактировать во время выполнения приложения, для них допустимо только свойство `Filter`.

При создании текста фильтра для свойства `Filter` указываются имена полей соответствующей таблицы БД, а для задания отношений применяются все операторы сравнения (>, >=, <, <=, =, <>) и логические операторы AND, OR, NOT:

```
Field1>100 AND Field2=20
```

Сравнивать между собой два поля нельзя. Например, применение фильтра

```
ItemCount=Balance AND InputPrice>OutputPrice
```

приведет к ошибке.

При создании динамических фильтров можно изменять как выражение фильтра целиком, так и его части. Например, ограничивающее значение для поля можно задавать при помощи элементов управления формы, что позволяет пользователю приложения управлять фильтрацией набора данных:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 with Table1 do
 begin
  Filtered := False;
  Filter := 'Field1>=' + TEdit(Sender).Text;
  Filtered := True;
 end;
end;
```

В фильтрах можно производить отбор по частям строк для строковых полей, для этого служит символ звездочка (*):

```
ItemName='A*'
```

Фильтр начинает работать только после того, как свойству `Filtered` присваивается истинное значение. Перед изменением текста динамического фильтра или для отключения фильтра свойству `Filtered` присваивается значение `False`.

Параметры фильтра определяются свойством `FilterOptions`:

```
property FilterOptions: TFilterOptions;
TFilterOption = (foCaseInsensitive, foNoPartialCompare);
TFilterOptions = set of TFilterOption;
```

Параметр `foCaseInsensitive`, будучи включенным в свойстве, отключает сравнение строковых значений с учетом регистра символов.

Параметр `foNoPartialCompare` отключает отбор строковых значений по части строки.

Метод-обработчик `OnFilterRecord` имеет следующее объявление:

```
type TFilterRecordEvent = procedure(DataSet: TDataSet; var Accept:
Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

Если этот метод создан для набора данных, то он вызывается для каждой его записи. Программный код метода должен присваивать параметру `Accept` истинное или ложное значение. В результате запись передается в набор данных или отсекается:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
 Accept := ArchOrdersArchDat.AsString >= DateEdit1.Text;
end;
```

Важнейшее преимущество метода `OnFilterRecord` по сравнению со свойством `Filter` в том, что в этом методе-обработчике можно сравнивать поля и производить вычисления над их значениями.

Недостатком метода является малая гибкость, хотя такой фильтр можно модифицировать путем присвоения методу процедурной переменной, содержащей ссылку на новый метод.

Быстрый переход к помеченным записям

Закладки, как инструмент работы с записями набора данных, позволяют осуществлять быстрое перемещение на нужную запись. Набор данных может содержать неограниченное число закладок, каждая из которых представляет собой указатель. Закладку можно создать только для текущей записи набора данных.

При работе с закладками используются три основные метода.

- `GetBookmark` — создает новую закладку для текущей записи.
- `GotoBookmark` — осуществляет переход к закладке, переданной в параметре.
- `FreeBookmark` — удаляет закладку, переданную в параметре.

Кроме этого, есть метод `BookmarkValid`, который проверяет, указывает ли закладка на реально существующую запись, и `CompareBookmark`, который позволяет сравнить между собой две закладки:

```
var Bookmark1, Bookmark2: TBookmark;
...
if Table1.CompareBookmark(Bookmark1, Bookmark2) = 1
then ShowMessage('Закладки одинаковы');
```

В наборе данных имеется свойство `Bookmark`, которое содержит название текущей закладки.

Рис. 18.4. Главная форма проекта DemoBookmark

Рассмотрим небольшой пример, где право управлять закладками предоставлено пользователю (рис. 18.4 и листинг 18.2). На форме, помимо других элементов управления (среди которых есть компонент `TDBGrid`), имеется две кнопки. Кнопка `StartBookmark` помечает текущую запись, кнопка `StopBookmark` переходит к закладке, а затем уничтожает ее.

Листинг 18.2. Пример использования закладок

```
implementation
{$R *.DFM}
```

```
var SaveRecPos: TBookMark;
procedure TMainForm.FormShow(Sender: TObject);
begin
 try
  Cust.Open;
  BookmarkControl.Brush.Color := clBtnFace;
 except
  ShowMessage('Ошибка открытия набора данных');
 end;
end;
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 Cust.Close;
end;
procedure TMainForm.StartBookmarkClick(Sender: TObject);
begin
 if Not Cust.BookmarkValid(SaveRecPos)
  then SaveRecPos := Cust.GetBookmark;
 BookmarkControl.Brush.Color := clLime
end;
procedure TMainForm.StopBookmarkClick(Sender: TObject);
begin
 with Cust do
 begin
  if Not BookmarkValid(SaveRecPos)
   then Exit;
  GotoBookmark(SaveRecPos);
  FreeBookmark(SaveRecPos);
  SaveRecPos := Nil;
 end;
 BookmarkControl.Brush.Color := clBtnFace;
end;
end.
```

Использование метода `BookmarkValid` позволяет корректно переопределять закладку, если она уже установлена, и избежать ошибок при произвольных нажатиях кнопок. Компонент `BookmarkControl` типа `TShape` сигнализирует о том, что закладка установлена или удалена.

Примечание

Закладки также используются в компоненте `TDBGrid`. Он имеет свойство `SelectedRows` типа `TBookmarkList`, которое представляет собой список закладок, указывающих на одновременно выделенные записи.

Диапазоны

В наборе данных, помимо фильтров, имеется еще одно средство отбора записей. Группа методов позволяет на основе индексов отбирать в набор данных только те записи, значения индексированных полей которых (для текущего индекса) соответствуют диапазону заданных величин.

Диапазоны работают быстрее фильтров, но менее гибки и не так удобны в работе. В случае применения диапазонов набор данных обязательно должен находиться в состоянии `dsSetKey`.

Для того чтобы включить диапазон, необходимо задать стартовое и конечное значение диапазона для ключевых полей, затем применить созданный диапазон к набору данных. Работающий диапазон можно модифицировать.

Примечание

Все методы работы с диапазонами используют те поля, которые заданы в текущем индексе. Для таблиц Paradox и dBASE это свойство `IndexName`. Для таблиц серверов SQL это свойство `IndexFieldNames`.

Метод `SetRangeStart` переводит набор данных в режим `dsSetKey`, это означает, что следующее за этим присваивание ключевым полям значений означает задание начальной границы диапазона.

Метод `SetRangeEnd` переводит набор данных в режим `dsSetKey`, следующее за этим присваивание ключевым полям значений означает задание конечной границы диапазона.

После этого необходимо вызвать метод `ApplyRange`, который применяет созданный диапазон к набору данных:

```
with Table1 do
begin
 SetRangeStart;
 Fields[0].Value := '439';
 SetRangeEnd;
 Fields[1].Value := '522';
 ApplyRange;
end;
```

Работающий диапазон можно модифицировать аналогичным образом: после вызова методов `EditRangeStart` и `EditRangeEnd` необходимо задать новые границы для ключевых полей и снова вызвать метод `ApplyRange`:

```
with Table1 do
begin
 EditRangeStart;
 Fields[0].Value := '500';
 EditRangeEnd;
 Fields[1].Value := '522';
 ApplyRange;
end;
```

Отмена диапазона осуществляется методом `CancelRange`.

Если индекс содержит несколько полей, то перед вызовом метода `ApplyRange` необходимо задать значения для всех ключевых полей.

С помощью метода `SetRange` можно одновременно задать верхнюю и нижнюю границы диапазона:

```
with Table1 do
begin
 SetRange(['500'], ['522']);
 ApplyRange;
end;
```

Тем, какая граница будет у диапазона — открытая или закрытая — управляет свойство `KeyExclusive`. Если оно имеет значение `True`, граничные значения в диапазон не включаются, в противном случае — включаются.

Резюме

В распоряжении разработчика приложений БД в Delphi имеется ряд полезных механизмов набора данных, которые реализованы для компонентов всех технологий доступа к данным.

К этим механизмам относятся методы быстрого поиска и перехода к найденным записям, связывания наборов данных по индексированным полям, а также дополнительной фильтрации записей набора данных.

Глава 19

Компоненты отображения данных

До этого момента мы рассматривали аспекты создания приложений баз данных, касающиеся организации доступа к данным и создания в приложениях наборов данных. Теперь более подробно остановимся на вопросах отображения данных в приложениях (интерфейс приложений).

Отображение данных обеспечивает достаточно представительный набор компонентов VCL Delphi. Многие из них унаследованы от компонентов, инкапсулирующих стандартные элементы управления. Для связи с набором данных эти компоненты используют компонент `TDataSource`.

Механизмы управления данными реализованы в компонентах наборов данных и активно взаимодействуют с компонентами отображения данных.

В этой главе рассматриваются следующие вопросы.

- Стандартные компоненты отображения данных.
- Навигация по данным.
- Механизм синхронного просмотра данных.
- Использование графиков для представления данных.

Классификация компонентов отображения данных

Все компоненты отображения данных можно разделить на группы по нескольким критериям (рис. 19.1).

Большинство компонентов предназначены для работы с отдельным полем, т. е. при перемещении по записям набора данных такие компоненты показывают текущие значения только одного поля. Для соединения с набором данных через компонент `TDataSource` предназначено свойство `DataSource`. Поле задается свойством `DataField`.

Компоненты TDBGrid и TDBCtrlGrid обеспечивают просмотр наборов данных целиком или в произвольном сочетании полей. В них присутствует только свойство DataSource.

Особенную роль среди компонентов отображения данных играет компонент TDBNavigator. Он не показывает данные и не предназначен для их редактирования, зато обеспечивает навигацию по набору данных.

Наиболее часто в практике программирования встречаются компоненты TDBGrid, TDBEdit и TDBNavigator.

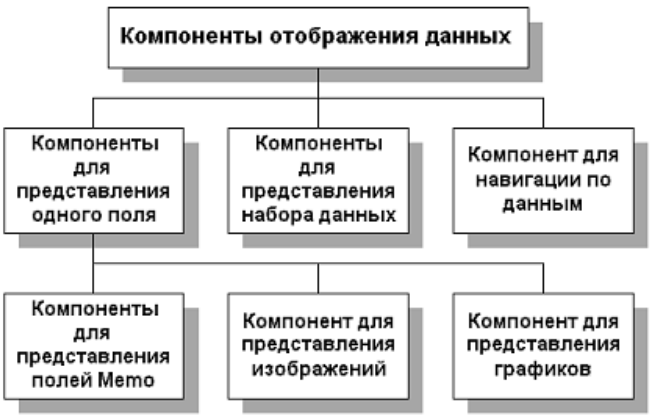


Рис. 19.1. Классификация компонентов отображения данных

Для представления и редактирования информации, содержащейся в полях типа Memo, предусмотрены специальные компоненты TDBMemo и TDBRichEdit.

Для просмотра (без редактирования) изображений предназначен компонент TDBImage.

Отдельную группу составляют компоненты синхронного просмотра данных. Они обеспечивают показ значений поля из одной таблицы в соответствии со значениями поля из другой таблицы.

Наконец, данные можно представить в виде графика. Для этого существует компонент TDBChart.

Как видите, набор компонентов отображения данных весьма разнообразен и позволяет решать задачи по созданию любых интерфейсов для приложений баз данных.

Ввиду общности решаемых задач компоненты отображения данных имеют несколько важных общих свойств, которые представлены в табл. 19.1 и в дальнейшем изложении опущены.

Таблица 19.1. *Общие свойства компонентов отображения данных*

Объявление	Описание
`property DataField: string;`	Поле связанного с компонентом набора данных
`property DataSource: TDataSource;`	Связываемый с компонентом компонент `TDataSource`
`property Field: TField;`	Обеспечивает доступ к классу `TField`, соответствующему полю набора данных, заданному свойством `DataField`
`property ReadOnly: Boolean;`	Управляет работой режима "только для чтения"

Табличное представление данных

Компонент *TDBGrid*

Этот компонент инкапсулирует двумерную таблицу, в которой строки представляют собой записи, а столбцы — поля набора данных.

Компонент `TDBGrid` является потомком классов `TDBCustomGrid` и `TCustomGrid`. От `TCustomGrid` наследуются все функции отображения и управления работой двумерной структуры данных. `TDBCustomGrid` обеспечивает визуализацию и редактирование полей из набора данных, причем `TDBGrid` только публикует свойства и методы класса `TDBCustomGrid`, не добавляя собственных.

В `TDBGrid` можно отображать произвольное подмножество полей текущего набора данных, но число записей ограничить нельзя — в компоненте всегда присутствуют все записи связанного набора данных. Требуемый набор полей можно составить при помощи специального Редактора столбцов, который открывается при двойном щелчке на компоненте, перенесенном на форму или кнопкой свойства `Columns` в Инспекторе объектов.

Новая колонка добавляется при помощи кнопки **Add New**, после этого ее название появляется в списке колонок (рис. 19.2). Для выбранной в списке колонки доступные для редактирования свойства появляются в Инспекторе объектов. Колонки в списке можно редактировать, удалять, менять местами.

При помощи кнопки **Add All Fields** в сетку можно добавить все поля набора данных.

Каждая колонка компонента `TDBGrid` описывается специальным классом `TColumn`, а совокупность колонок доступна через свойство `Columns` компонента, оно имеет тип `TDBGridColumns` и представляет собой индексирован-

ный список объектов колонок. Поле набора данных связывается с конкретной колонкой при помощи свойства `FieldName` класса `TColumn`. При этом в колонку автоматически переносятся все необходимые параметры поля, в частности, заголовок поля, настройки шрифтов, ширина поля. После ручного изменения параметров первоначальные значения восстанавливаются методами соответствующих объектов `TColumn`.

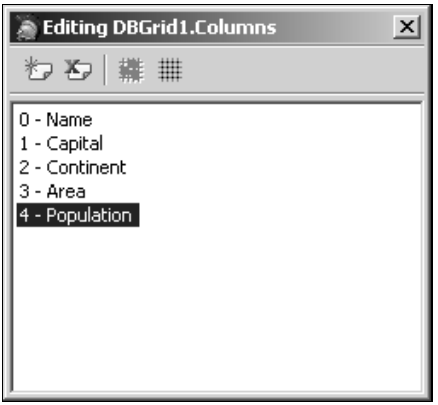


Рис. 19.2. Редактор колонок компонента `TDBGrid`

При помощи метода `DefaultDrawColumnCell` и метода-обработчика `OnDrawColumnCell` можно управлять процессом отображения данных в ячейках.

Метод `DefaultDrawDataCell` предназначен только для обеспечения обратной совместимости по коду с более ранними версиями.

Настройка параметров компонента `TDBGrid`, от которых зависят его внешний вид и некоторые функции, осуществляется при помощи свойства `Options` (табл. 19.2). Текущая позиция в двумерной структуре данных может быть определена свойствами `SelectedField`, `SelectedRows`, `SelectedIndex`.

При необходимости программист может выбрать разнообразные методы-обработчики событий. Среди них есть как стандартные методы, присущие всем элементам управления, так и специфические.

Например, при помощи метода-обработчика `OnEditButtonClick` можно предусмотреть вызов специализированной формы при щелчке на кнопке в ячейке:

```
procedure TForm1.DBGrid1EditButtonClick(Sender: TObject);
begin
 if DBGrid1.SelectedIndex = 2 then SomeForm.ShowModal;
end;
```

Примечание

Объект колонки `TColumn` имеет свойство `ButtonStyle`. Если ему присвоить значение `cbsEllipsis`, то при активизации ячейки этой колонки в правой части ячейки появляется кнопка.

Таблица 19.2. *Свойства и методы компонента* `TDBGrid`

Объявление	Тип	Описание
Свойства		
`property Columns: TDBGridColumns;`	Pb	Содержит коллекцию объектов `TColumn`, описывающих колонки компонента
`property DefaultDrawing: Boolean;`	Pb	Определяет способ визуализации данных в сетке. При значении `True` данные отображаются автоматически. При значении `False` используется метод-обработчик `OnDrawColumnCell`
`property FieldCount: Integer;`	Ro	Возвращает число видимых колонок сетки
`property Fields[Index: Integer]: TField;`	Ro	Массив объектов полей набора данных, отображаемых в компоненте
`TDBGridOption = (dgEditing, dgAlwaysShowEditor, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgRowSelect, dgAlwaysShowSelection, dgConfirmDelete, dgCancelOnExit, dgMultiSelect);` `TDBGridOptions = set of TDBGridOption;`	Pb	Определяет особенности визуализации и поведения компонента. `DgEditing` — данные можно редактировать. `DgAlwaysShowEditor` — данные в сетке всегда в режиме редактирования. `dgTitles` — видны заголовки колонок. `dgIndicator` — в начале строки виден номер текущей колонки. `dgColumnResize` — колонки можно перемещать и менять их ширину. `dgColLines` — видны линии между колонками. `dgRowLines` — видны линии между строками. `dgTabs` — для перемещения по строкам можно использовать клавиши <Tab> и <Shift>+<Tab>. `dgRowSelect` — можно выделять целые строки, при этом игнорируются установки `dgEditing` и `dgAlwaysShowEditor`.

Таблица 19.2 *(продолжение)*

Объявление	Тип	Описание
Свойства		
		`DgAlwaysShowSelection` — выделение текущей ячейки сохраняется, даже если сетка не активна. `DgConfirmDelete` — при удалении строк появляется запрос о подтверждении операции. `DgCancelOnExit` — созданные пустые строки при уходе из сетки не сохраняются. `DgMultiSelect` — можно выделять несколько строк одновременно
`property SelectedField: TField;`	Pu	Содержит объект текущего поля
`property SelectedIndex: Integer;`	Pu	Содержит номер текущей колонки в массиве свойства `Columns`
`property SelectedRows: TBookmarkList;`	Ro	Набор закладок на записи набора данных, соответствующих выделенным строкам сетки
`property TitleFont: TFont;`	Pb	Шрифт заголовков колонок
`property EditorMode: Boolean;`	Pu	Показывает, можно ли редактировать текущую ячейку.
`property FixedColor: TColor;`	Pb	Цвет фона неподвижных ячеек сетки
Методы		
`procedure DefaultDrawColumnCell (const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState);`	Pu	Перерисовывает текст в ячейке колонки с номером `DataCol`. Ячейка задается прямоугольником `Rect` на канве сетки. Параметр `State` определяет состояние ячейки после перерисовки. Параметр `Column` содержит экземпляр класса колонки, которой принадлежит ячейка
`procedure DefaultDrawDataCell (const Rect: TRect; Field: TField; State: TGridDrawState);`	Pu	Перерисовывает текст в ячейке колонки, определяемой параметром `Field`, содержащим связанный с колонкой объект поля. Ячейка задается прямоугольником `Rect` на канве сетки. Параметр `State` определяет состояние ячейки после перерисовки

Таблица 19.2 *(продолжение)*

Объявление	Тип	Описание
Методы		
`procedure DefaultHandler (var Msg); override;`	Pu	Вызывает всплывающее меню для колонки, которой соответствуют текущие координаты мыши. Компонент должен обрабатывать сообщение `WM_RBUTTONUP`
`function ExecuteAction(Action: TBasicAction): Boolean; override;`	Pu	Выполняет действие, заданное параметром `Action`, по отношению к данному компоненту
`procedure ShowPopupEditor (Column: TColumn; X: Integer = Low(Integer); Y: Integer = Low(Integer)); dynamic;`	Pu	Открывает набор данных, связанный с передаваемой параметром `Column` колонкой в новом окне. Работает только для типов данных "абстрактный" и "набор данных". Параметры `X` и `Y` определяют положение нового окна
`function ValidFieldIndex (FieldIndex: Integer): Boolean;`	Pu	Возвращает `True`, если колонка с номером `FieldIndex` связана с полем набора данных

`type TGridCoord = record`
`X: Longint;`
`Y: Longint;`
`end;`
`function MouseCoord(X, Y: Integer): TGridCoord;`
	Pu	Возвращает номера строки и столбца, соответствующих ячейке, которой принадлежат экранные координаты `X` и `Y`

Методы-обработчики событий		
`type TDBGridClickEvent = procedure (Column: TColumn) of object;` `property OnCellClick: TDBGridClickEvent;`	Pb	Вызывается при щелчке мышью на ячейке. Параметр `Column` содержит колонку, которой принадлежит ячейка
`property OnColEnter: TNotifyEvent;`	Pb	Вызывается при переносе фокуса на новую колонку сетки
`property OnColExit: TNotifyEvent;`	Pb	Вызывается перед переносом фокуса из текущей колонки

Таблица 19.2 *(окончание)*

Объявление	Тип	Описание
Методы-обработчики событий		
`type TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: Longint) of object;` `property OnColumnMoved: TMovedEvent;`	Pb	Вызывается при переносе колонки в сетке на новое место при помощи мыши. Параметр `FromIndex` возвращает номер старого положения колонки. Параметр `ToIndex` возвращает номер нового положения колонки

`type`
`TDrawColumnCellEvent = procedure (Sender: TObject; const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState) of object;`
`property OnDrawColumnCell: TDrawColumnCellEvent;`
	Pb	Вызывается при перерисовке ячейки.
		`Rect` — определяет ячейку по координатам прямоугольника на канве.
		`DataCol` — возвращает номер колонки в сетке.
		`Column` — содержит объект колонки.
		`State` — возвращает состояние колонки

`type`
`TDrawDataCellEvent = procedure (Sender: TObject; const Rect: TRect; Field: TField; State: TGridDrawState) of object;`
`property OnDrawDataCell: TDrawDataCellEvent;`
	Pb	Вызывается при перерисовке ячейки, перед обработчиком `OnDrawColumnCell`, если свойство `Columns.State = csDefault`.
		Этот метод лучше не применять, т. к. он используется только для обеспечения обратной совместимости с ранними версиями

`property OnEditButtonClick: TNotifyEvent;`	Pb	Вызывается при щелчке мышью на кнопке в ячейке
`type TDBGridClickEvent = procedure (Column: TColumn) of object;` `property OnTitleClick: TDBGridClickEvent;`	Pb	Вызывается при щелчке мышью на заголовке колонки. Колонка определяется параметром `Column`

В работе компонента TDBGrid важную роль играет класс TColumn, который инкапсулирует свойства колонки или столбца сетки. Его основным назначением является правильное отображение данных из поля набора данных, связанного с этой колонкой. Поэтому объект колонки обладает свойствами и методами, которые позволяют произвольным образом задавать параметры отображения данных (цвет, шрифт, ширину и т. д.). Первоначальные значения берутся из связанных с колонками полей. Измененные свойства можно восстановить при помощи группы специальных методов (DefaultColor, DefaultFont и др.).

Свойство AssignedValues позволяет в любой момент определить, какие первоначальные настройки были изменены.

За отображение заголовка колонки отвечает свойство Title, представляющее собой ссылку на экземпляр объекта TColumnTitle. Здесь можно задать текст заголовка, параметры шрифта текста заголовка и цвет фона заголовка. По умолчанию текст заголовка берется из свойства DisplayLabel объекта TField *(см. главу 17)*.

Каждой колонке можно придать список, который разворачивается при щелчке на кнопке в активной ячейке колонки. Выбранное в списке значение автоматически заносится в ячейку. Для реализации этой возможности применяется свойство PickList типа TStrings. Достаточно лишь заполнить список значениями во время разработки или выполнения (рис. 19.3). Свойства и методы класса TColumn приведены в табл. 19.3.

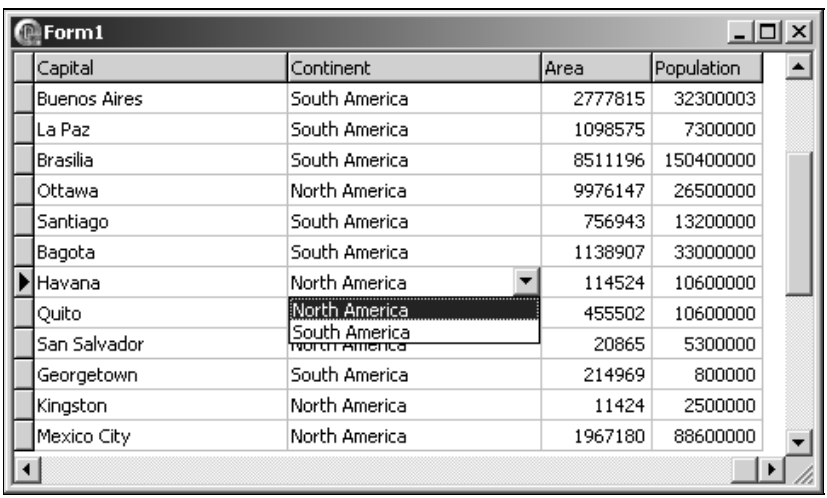


Рис. 19.3. Список колонки в компоненте TDBGrid

Таблица 19.3. *Свойства и методы класса* `TColumn`

Объявление	Тип	Описание
Свойства		
`property Alignment: TAlignment;`	Pb	Определяет выравнивание данных в колонке

`type`
`TColumnValue = (cvColor, cvWidth, cvFont, cvAlignment, cvReadOnly, cvTitleColor, cvTitleCaption, cvTitleAlignment, cvTitleFont, cvImeMode, cvImeName);`
`TColumnValues = set of TColumnValue;`
`property AssignedValues: TColumnValues;`
	Ro	Возвращает набор атрибутов колонки, которые были изменены по сравнению с первоначальными

`type TColumnButtonStyle = (cbsAuto, cbsEllipsis, cbsNone);` `property ButtonStyle: TColumnButtonStyle;`	Pb	Задает способ редактирования данных в колонке. `cbsAuto` — кнопка в редактируемой ячейке появляется, если связанное поле является полем синхронного просмотра. `cbsEllipsis` — кнопка в редактируемой ячейке появляется всегда, щелчок на кнопке вызывает обработчик `OnEditButtonClick`. `cbsNone` — при редактировании ячейки кнопка не появляется
`property Color: TColor;`	Pb	Цвет фона колонки
`property DisplayName: string;`	Pu	Название колонки в списке Редактора столбцов
`property DropDownRows: Cardinal;`	Pb	Определяет число строк разворачивающегося списка ячейки
`property Expandable: Boolean;`	Pu	В значении `True` разрешает показ связанных с полем колонки дочерних полей абстрактного, ссылочного типов и массивов

Таблица 19.3 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`property Expanded: Boolean;`	Pb	При значении `True` каждое дочернее поле отображается в новой колонке. При значении `False` дочерние поля отображаются через точку с запятой и не доступны для редактирования
`property FieldName: string;`	Pb	Название поля, связанного с колонкой
`property Font: TFont;`	Pb	Шрифт данных в колонке
`property Grid: TCustomDBGrid;`	Ro	Определяет сетку, содержащую эту колонку
`property ParentColumn: TColumn;`	Ro	Определяет колонку — владельца текущей колонки. Используется для дочерних полей
`property PickList: TStrings;`	Pb	Содержит разворачивающийся список, используемый при редактировании данных
`property PopupMenu: TPopupMenu;`	Pb	Связывает с колонкой всплывающее меню
`property Showing: Boolean;`	Ro	Возвращает `True`, если колонка видима
`property Title: TColumnTitle;`	Pb	Задает текст заголовка и его параметры
`property Visible: Boolean;`	Pb	Задает видимость колонки
`property Width: Integer;`	Pb	Задает ширину колонки в пикселах
Методы		
`procedure Assign(Source: TPersistent); override;`	Pu	Копирует колонку `Source` в текущую колонку
`function DefaultAlignment: TAlignment;`	Pu	Возвращает первоначальное значение выравнивания колонки
`function DefaultColor: TColor;`	Pu	Возвращает первоначальный фоновый цвет колонки
`function DefaultFont: TFont;`	Pu	Возвращает первоначальный шрифт данных в колонке

Таблица 19.3 *(окончание)*

Объявление	Тип	Описание
Методы		
`type TImeMode = (imDisable, imClose, imOpen, imDontCare, imSAlpha, imAlpha, imHira, imSKata, imKata, imChinese, imSHanguel, imHanguel);` `function DefaultImeMode: TImeMode;`	Pu	Возвращает первоначальный способ ввода символов
`type TImeName = type string;` `function DefaultImeName: TImeName;`	Pu	Возвращает первоначальное имя редактора способа ввода символов
`function DefaultReadOnly: Boolean;`	Pu	Возвращает первоначальный режим редактирования данных
`function DefaultWidth: Integer;`	Pu	Возвращает первоначальную ширину колонки в пикселах
`function Depth: Integer;`	Pu	Возвращает число непосредственных предков колонки
`procedure RestoreDefaults; virtual;`	Pu	Восстанавливает первоначальные настройки колонки

При работе с компонентом `TDBGrid` все операции с отдельными колонками осуществляются при помощи экземпляра класса `TDBGridColumns`, который инкапсулирует список объектов колонок (свойство `Columns` компонента `TDBGrid`). Доступ к колонкам осуществляется при помощи свойства `Items`. Нумерация колонок начинается с нуля.

При помощи свойств и методов класса `TDBGridColumns` можно изменять настройки полей компонента `TDBGrid` во время выполнения.

Свойство `State` определяет способ создания колонок. Его значение устанавливается автоматически. При создании колонок для всех полей сразу (кнопка **Add All Fields** Редактора столбцов) устанавливается значение `csDefault`. При любом ручном изменении свойств устанавливается значение `csCustomized`. При программном изменении значения свойства во время выполнения все существующие колонки удаляются.

Все данные из существующих колонок можно сохранить в файле или потоке при помощи методов `SaveToFile` и `SaveToStream`, а затем загрузить их обратно методами `LoadFromFile` и `LoadFromStream`.

Свойства и методы класса `TDBGridColumns` приведены в табл. 19.4.

Таблица 19.4. *Свойства и методы класса* `TDBGridColumns`

Объявление	Тип	Описание
Свойства		
`property Grid: TCustomDBGrid;`	Ro	Возвращает ссылку на сетку, владеющую данным объектом
`property Items[Index: Integer]: TColumn default;`	Pu	Индексный список объектов колонок сетки
`type TDBGridColumnsState = (csDefault, csCustomized);` `property State: TDBGridColumnsState;`	Pu	Определяет способ создания колонок сетки. `csDefault` — колонки создаются динамически, с параметрами, соответствующими связанным полям. `csCustomized` — параметры колонок определены разработчиком и могут отличаться от параметров полей
`property Count: Integer;`	Pu	Возвращает общее число колонок
Методы		
`function Add: TColumn;`	Pu	Добавляет новый объект `TColumn`
`procedure LoadFromFile(const Filename: string);`	Pu	Загружает данные в объект из файла `FileName`
`procedure LoadFromStream(S: TStream);`	Pu	Загружает данные в объект из потока `S`
`procedure RebuildColumns;`	Pu	Удаляет существующие колонки и создает новые, основываясь на параметрах полей набора данных
`procedure RestoreDefaults;`	Pu	Восстанавливает первоначальные настройки колонок
`procedure SaveToFile(const Filename: string);`	Pu	Сохраняет данные из колонок в файле `FileName`
`procedure SaveToStream(S: TStream);`	Pu	Сохраняет данные из колонок в потоке `S`

Компонент *TDBCtrlGrid*

Компонент TDBCtrlGrid внешне напоминает компонент TDBGrid, но никак не связан с классом TCustomDBGrid, а наследует напрямую от класса TWinControl.

Этот компонент позволяет отображать данные в строках в произвольной форме. Компонент представляет собой набор панелей, каждая из которых служит платформой для размещения данных отдельной записи набора данных. На панели могут размещаться любые компоненты отображения данных, предназначенные для работы с отдельным полем. С каждым таким компонентом можно связать нужное поле набора данных. При открытии набора данных в компоненте TDBCtrlGrid на каждой новой панели создается набор компонентов отображения данных, аналогичный тому, который был создан на одной панели во время разработки.

На панель можно переносить только те компоненты отображения данных, которые показывают значение одного поля для единственной записи набора данных. Нельзя использовать компоненты TDBGrid, TDBCtrlGrid, TDBRichEdit, TDBListBox, TDBRadioGroup и TDBLookupListBox.

После того как для компонента TDBCtrlGrid задано значение свойства DataSource, все переносимые на панель компоненты отображения данных автоматически связываются с указанным компонентом TDataSource. Самостоятельное задание свойства DataSource для дочерних компонентов отображения данных не допускается. В них требуется определить только поля.

Компонент может отображать панели в одну или несколько колонок. Для задания числа колонок панелей используется свойство ColCount. Число видимых строк панелей определяется свойством RowCount. Вертикальное или горизонтальное размещение колонок панелей зависит от значения свойства Orientation.

При наличии нескольких колонок панелей курсор перемещается по колонке сверху вниз с последующим переходом на следующую колонку. Направление движения курсора не зависит от значения свойства Orientation.

Размеры одной панели определяются свойствами PanelHeight и PanelWidth. Они взаимосвязаны с размерами самого компонента. При изменении значений свойств PanelHeight и PanelWidth размеры компонента изменяются таким образом, чтобы в нем помещалось указанное в свойствах ColCount и RowCount число панелей и наоборот.

Не рекомендуется размещать на панели компоненты TDBMemo и TDBImage, т. к. это может привести к значительному снижению производительности.

Свойства и методы компонента TDBCtrlGrid приведены в табл. 19.5.

Таблица 19.5. *Свойства и методы компонента* `TDBCtrlGrid`

Объявление	Тип	Описание
Свойства		
`property AllowDelete: Boolean;`	Pb	Разрешает или запрещает удаление текущей записи
`property AllowInsert: Boolean;`	Pb	Разрешает или запрещает вставку новой записи
`property Canvas: TCanvas;`	Ro	Канва компонента
`property ColCount: Integer;`	Pb	Определяет число колонок с панелями
`property EditMode: Boolean;`	Pu	Разрешает или запрещает редактирование данных
`type TDBCtrlGridOrientation = (goVertical, goHorizontal);` `property Orientation: TDBCtrlGridOrientation;`	Pb	Определяет порядок следования записей: по горизонтали или по вертикали
`type TDBCtrlGridBorder = (gbNone, gbRaised);` `property PanelBorder: TDBCtrlGridBorder;`	Pb	Определяет способ отображения границы панели
`property PanelCount: Integer;`	Ro	Содержит число видимых одновременно панелей
`property PanelHeight: Integer;`	Pb	Определяет высоту панелей в пикселах
`property PanelIndex: Integer;`	Pu	Определяет индекс панели текущей записи
`property PanelWidth: Integer;`	Pb	Определяет ширину панелей в пикселах
`property RowCount: Integer;`	Pb	Определяет число строк видимых панелей
`property SelectedColor: TColor;`	Pb	Определяет фоновый цвет панели текущей записи
`property ShowFocus: Boolean;`	Pb	Разрешает или запрещает выделение вокруг панели текущей записи

Таблица 19.5 *(окончание)*

Объявление	Тип	Описание
Методы		
`type TDBCtrlGridKey = (gkNull, gkEditMode, gkPriorTab, gkNextTab, gkLeft, gkRight, gkUp, gkDown, gkScrollUp, gkScrollDown, gkPageUp, gkPageDown, gkHome, gkEnd, gkInsert, gkAppend, gkDelete, gkCancel);` `procedure DoKey(Key: TDBCtrlGridKey);`		Выполняет операцию, заданную при помощи параметра `Key`. Доступны операции навигации по записям, перевода в режим редактирования, вставки, удаления записей, отмены изменений
`procedure KeyDown(var Key: Word; Shift: TShiftState); override;`		Используется при нажатии клавиши для трансляции их кодов
Методы-обработчики событий		
`type TPaintPanelEvent = procedure(DBCtrlGrid: TDBCtrlGrid; Index: Integer) of object;` `property OnPaintPanel: TPaintPanelEvent;`		Вызывается при перерисовке панели. Параметр `Index` соответствует индексу панели

Навигация по набору данных

Перемещение или навигация по записям набора данных может осуществляться несколькими путями. Например, в компонентах `TDBGrid` и `TDBCtrlGrid`, которые отображают сразу несколько записей набора данных, можно использовать клавиши вертикального перемещения курсора или вертикальную полосу прокрутки.

Но что делать, если на форме расположены только компоненты, отображающие одно поле только текущей записи набора данных (`TDBEdit`, `TDBComboBox` и т. д.)? Очевидно, что в этом случае на форме должны быть расположены дополнительные элементы управления, отвечающие за перемещение по записям.

Аналогично, ни один компонент отображения данных не имеет встроенных средств для создания и удаления записей целиком.

Для решения указанных задач и предназначен компонент `TDBNavigator`, который представляет собой совокупность управляющих кнопок, выполняет операции навигации по набору данных и модификации записей целиком.

Компонент `TDBNavigator` при помощи свойства `DataSource` связывается с компонентом `TDataSource` и через него с набором данных. Такая схема позволяет обеспечить изменение текущих значений полей сразу во всех связанных с `TDataSource` компонентах отображения данных. Таким образом, `TDBNavigator` только дает команду на выполнение перемещения по набору данных или другой управляющей операции, а всю реальную работу выполняют компонент набора данных и компонент `TDataSource`. Компонентам отображения данных остается только принять новые данные от своих полей.

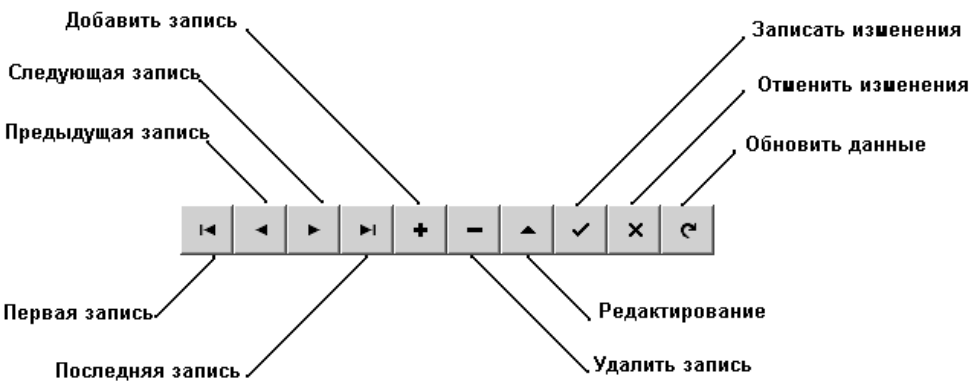


Рис. 19.4. Назначение кнопок компонента `TDBNavigator`

Компонент `TDBNavigator` содержит набор кнопок, каждая из которых отвечает за выполнение одной операции над набором данных (рис. 19.4). Всего имеется 10 кнопок, разработчик может оставить в наборе любое количество кнопок в любом сочетании. Видимостью кнопок управляет свойство `VisibleButtons`:

```
type
  TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete,
nbEdit, nbPost, nbCancel, nbRefresh);
  TButtonSet = set of TNavigateBtn;
property VisibleButtons: TButtonSet;
```

Каждый элемент типа `TNavigateBtn` представляет одну кнопку. Далее перечислено их назначение:

- `nbFirst` — перемещение на первую запись набора данных;
- `nbPrior` — перемещение на предыдущую запись набора данных;
- `nbNext` — перемещение на следующую запись набора данных;
- `nbLast` — перемещение на последнюю запись набора данных;
- `nbInsert` — вставка новой записи в текущей позиции набора данных;
- `nbDelete` — удаление текущей записи, курсор перемещается на следующую запись;
- `nbEdit` — набор данных переводится в режим редактирования;
- `nbPost` — в базу данных переносятся все изменения в текущей записи;
- `nbCancel` — все изменения в текущей записи отменяются;
- `nbRefresh` — восстанавливаются первоначальные значения текущей записи, сделанные после последнего переноса изменений в базу данных.

Самой критичной к возможной потере данных вследствие ошибки является операция удаления записи, поэтому при помощи свойства `ConfirmDelete` можно включить механизм контроля удаления. При каждом удалении записи нужно будет дать подтверждение выполняемой операции.

Нажатие любой кнопки можно эмулировать программно при помощи метода `BtnClick`.

В случае необходимости выполнения дополнительных действий при щелчке на любой кнопке можно воспользоваться обработчиками событий `BeforeAction` и `OnClick`, в которых параметр `Button` определяет нажатую кнопку.

Свойства и методы компонента `TDBNavigator` приведены в табл. 19.6.

Таблица 19.6. *Свойства и методы компонента* `TDBNavigator`

Объявление	Тип	Описание
Свойства		
`property ConfirmDelete: Boolean;`	Pb	Включает или отключает подтверждение удаления записи
`property Hints: TStrings;`	Pb	Содержит список подсказок для каждой кнопки
`property Flat: Boolean;`	Pb	Определяет внешний вид кнопок компонента

Таблица 19.6 *(окончание)*

Объявление	Тип	Описание

`type`
`TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit, nbPost, nbCancel, nbRefresh);`
`TButtonSet = set of TNavigateBtn;`
`property VisibleButtons: TButtonSet;`
	Pb	Список видимых кнопок

Методы		
`procedure BtnClick(Index: TNavigateBtn);`	Pu	Эмулирует щелчок на кнопке `Index`
`procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer);`	Pu	Задает положение (параметры `ALeft`, `ATop`) и размер компонента (параметры `AWidth`, `AHeight`)
Методы-обработчики событий		
`ENavClick = procedure (Sender: TObject; Button: TNavigateBtn) of object;` `property BeforeAction: ENavClick;`	Pb	Выполняется при щелчке на кнопке `Button`, перед выполнением операции, связанной с кнопкой
`ENavClick = procedure (Sender: TObject; Button: TNavigateBtn) of object;` `property OnClick: ENavClick;`	Pb	Выполняется при щелчке на кнопке `Button` после выполнения операции, связанной с кнопкой

Представление отдельных полей

Большинство компонентов отображения данных предназначено для представления данных из отдельных полей. Для этого все они имеют свойство `DataField`, которое указывает на требуемое поле набора данных.

В зависимости от типа данных поля могут использоваться различные компоненты. Для большинства стандартных полей предусмотрены компоненты TDBText, TDBEdit, TDBComboBox и TDBListBox.

Данные в формате Memo отображаются компонентами TDBMemo и TDBRichEdit.

Для показа изображений предназначен компонент TDBImage.

Компонент *TDBText*

Этот компонент представляет собой статический текст, который отображает текущее значение некоторого поля связанного набора данных. При этом данные можно просматривать в режиме "только для чтения".

Непосредственным предком компонента является класс TCustomLabel, поэтому он очень похож на компонент TLabel.

При работе с компонентом следует обратить внимание на возможную длину отображаемых данных. Для предотвращения обрезания текста можно задействовать свойства AutoSize и WordWrap.

Компонент *TDBEdit*

Компонент представляет собой стандартный однострочный текстовый редактор, в котором отображаются и изменяются данные из поля связанного набора данных.

Прямой предок компонента — класс TCustomMaskEdit, который также является прямым предком компонента TEdit.

Компонент может осуществлять проверку редактируемых данных по заданной для поля маской. Непосредственно для редактора задать маску нельзя, т. к. содержащее маску свойство EditMask в классе TCustomMaskEdit является защищенным, а в TDBEdit не перекрыто. Тем не менее механизм контроля полностью унаследован. Саму же маску можно задать в связанном с редактором поле. Объект TField имеет собственное свойство EditMask, которое действует при проверке данных в редакторе *(см. главу 17)*.

Проверка редактируемого текста на соответствие маске осуществляется методом ValidateEdit после каждого введенного или измененного символа. В случае ошибки генерируется исключение ValidateError и курсор устанавливается на первый ошибочный символ.

В компоненте можно работать с буфером обмена. Это делается средствами операционной системы пользователем или программно при помощи методов CopyToClipboard, CutToClipboard и PasteFromClipboard.

Компонент *TDBCheckBox*

Компонент представляет собой почти полный аналог обычного флажка (компонент `TCheckBox`) и предназначен для отображения и редактирования любых данных, которые могут иметь только два значения. Это может быть логический тип данных или любые строковые значения, но поле может принимать значения только из двух строк.

Предопределенные значения задаются свойствами `ValueChecked` и `ValueUnchecked`. По умолчанию они имеют значения `True` и `False`. Этим свойствам можно также присваивать любые строковые значения, причем одному свойству можно назначить несколько возможных значений, разделенных точкой с запятой.

Включение флажка происходит, если значение поля набора данных совпадает со значением свойства `ValueChecked` (единственным или любым из списка). Если же флажок включил пользователь, то значение поля данных приравнивается к единственному или первому в списке значению свойства `ValueChecked`.

Аналогичные действия происходят и со свойством `ValueUnchecked`.

Компонент *TDBRadioGroup*

Компонент представляет собой стандартную группу переключателей, состояние которых зависит от значений поля связанного набора данных. В поле можно передавать фиксированные значения, связанные с отдельными переключателями в группе.

Если текущее значение связанного поля соответствует значению какого-либо переключателя, то он включается. Если пользователь включает другой переключатель, то связанное с переключателем значение заносится в поле.

Возможные значения, на которые должны реагировать переключатели в группе, заносятся в свойство `Values` при помощи специального редактора в Инспекторе объектов или программно при помощи методов класса `TStrings`. Каждому элементу свойства `Values` соответствует один переключатель (порядок следования сохраняется).

Свойство `Items` содержит список поясняющих надписей для переключателей группы. Если для какого-либо переключателя нет заданного значения, но есть поясняющий текст, то такой переключатель включается при совпадении значения связанного поля с поясняющим текстом.

Текущее значение связанного поля содержится в поле `Value`.

Компонент *TDBListBox*

Компонент отображает текущее значение связанного с ним поля набора данных и позволяет изменить его на любое фиксированное из списка. Функционально компонент ничем не отличается от компонента `TListBox`. Значение поля должно совпадать с одним из элементов списка.

Специальных методов компонент не содержит.

Компонент *TDBComboBox*

Компонент отображает текущее значение связанного с ним поля набора данных в строке редактирования, при этом значение поля должно совпадать с одним из элементов разворачивающегося списка. Текущее значение можно изменить на любое фиксированное из списка компонента. Функционально компонент ничем не отличается от компонента `TComboBox`, представляющего собой комбинированный список.

Компонент может работать в пяти различных стилях, которые определяются свойством `Style`.

Специальных методов компонент не содержит.

Компонент *TDBMemo*

Компонент представляет собой обычное поле редактирования, к которому подключается поле с типом данных `Memo` или BLOB. Основное его преимущество — возможность одновременного просмотра и редактирования нескольких строк переменной длины. Компонент может отображать только строки, которые целиком видны по высоте.

В компоненте можно использовать буфер обмена при помощи стандартных средств операционной системы или унаследованными от предка `TCustomMemo` методами `CopyToClipBoard`, `CutToClipBoard` и `PasteFromClipBoard`.

Для ускорения навигации по набору данных при отображении полей типа BLOB можно использовать свойство `AutoDisplay`. При значении `True` любое новое значение поля автоматически отображается в компоненте. При значении `False` новое значение появляется только после двойного щелчка на компоненте или после нажатия клавиши <Enter> при активном компоненте.

Метод `LoadMemo` используется автоматически при загрузке значения поля, если свойство `AutoDisplay = False`.

Поведением компонента при работе со слишком длинными строками можно управлять при помощи свойства `WordWrap`. При значении `True` слишком длинная строка сдвигается влево при перемещении текстового курсора за

правую границу компонента. При значении `False` остаток длинной строки переносится на новую строку, при этом реально новая строка в данных не создается.

Компонент *TDBImage*

Компонент предназначен для просмотра изображений, хранящихся в базах данных в графическом формате.

Редактировать изображения можно только в каком-либо графическом редакторе, перенося исходное и измененное изображения при помощи буфера обмена. Это делается средствами операционной системы пользователем или программно при помощи методов `CopyToClipboard`, `CutToClipboard` и `PasteFromClipboard`.

Визуализация изображения осуществляется при помощи свойства `Picture`, которое представляет собой экземпляр класса `TPicture`.

Также можно полностью заменить существующее изображение или сохранить новое в новой записи набора данных. Для этого предназначены методы свойства `Picture`.

Свойство `AutoDisplay` позволяет управлять процессом загрузки новых изображений из набора данных в компонент. При значении `True` любое новое значение поля автоматически отображается в компоненте. При значении `False` новое значение появляется только после двойного щелчка на компоненте или после нажатия клавиши <Enter> при активном компоненте.

Для ускорения просмотра изображений можно применить свойство `QuickDraw`, которое задает используемую изображением палитру. При значении `True` устанавливается стандартная системная палитра. В результате уменьшается время загрузки изображения, но может ухудшиться и качество изображения, в некоторых случаях до полного искажения. При значении `False` выбирается собственная палитра изображения и процесс загрузки замедляется.

Компонент *TDBRichEdit*

Компонент предоставляет возможности полноценного текстового редактора для просмотра и изменения текстовых данных, хранящихся в связанном поле набора данных. Поле должно содержать информацию о форматировании текста.

Внешне компонент ничем не отличается от поля редактирования, поэтому о реализации доступа к гораздо более богатым возможностям редактора через интерфейс пользователя должен позаботиться разработчик. Для этого существуют дополнительные элементы управления.

Синхронный просмотр данных

При разработке приложений для работы с базами данных часто возникает необходимость в связывании двух наборов данных по ключевому полю. Например, в таблице `Orders` (содержит данные о заказах) демонстрационной базы данных DBDEMOS имеется поле `CustNo`, которое содержит идентификационный номер покупателя. Под этим же номером в таблице `Customers` хранится информация о покупателе (адрес, телефон, реквизиты и т. д.). При разработке пользовательского интерфейса приложения баз данных необходимо, чтобы при просмотре перечня заказов в форме приложения отображались не идентификационные номера покупателей, а их параметры.

Таким образом, в наборе данных заказов вместо поля номера покупателя должно появиться поле имени покупателя из таблицы `Customers`. Механизм связывания полей из различных наборов данных по ключевому полю называется синхронным просмотром. В рассмотренном примере ключевым является поле `CustNo` из таблицы `Customers`, а конкретное наименование выбирается по совпадению значений ключевого поля и заменяемого поля из исходного набора данных `Orders`. Причем необходимо, чтобы в таблице `Customers` поле `CustNo` было уникальным (составляло первичный или вторичный ключ).

Таблицу, в которой расположено поле, значения которого замещаются на синхронные, будем называть *исходной таблицей* (это таблица `Orders`).

Таблицу, содержащую ключевое поле и поле данных для синхронного просмотра, будем называть *таблицей синхронного просмотра* (таблица `Customers`).

В Delphi механизм синхронного просмотра реализован на уровне отдельных полей и компонентов.

В наборе данных динамически можно создать специальное поле синхронного просмотра, которое будет автоматически замещать одно значение другим в зависимости от значения ключевого поля. Такое поле можно связать с любым рассмотренным ранее компонентом отображения данных *(см. главу 17)*.

Помимо простого синхронного просмотра данных может возникнуть задача редактирования данных в аналогичной ситуации. Для этого предназначены специальные компоненты синхронного просмотра данных, которые позволяют, например, выбирать покупателя из списка, а изменится при этом номер покупателя в наборе данных заказов. Использование таких компонентов делает пользовательский интерфейс значительно более удобным и наглядным. В VLC Delphi есть два таких компонента: `TDBLookupListBox` и `TDBLookupComboBox`.

Механизм синхронного просмотра

Непосредственным предком компонентов синхронного просмотра данных является класс `TDBLookupControl`, который инкапсулирует список значений для просмотра и сам механизм синхронного просмотра.

Как и в любом другом компоненте отображения данных, в компонентах синхронного просмотра должны присутствовать средства связывания с требуемым полем некоторого набора данных. Это уже известные свойства `DataSource` — для задания набора данных через компонент `TDataSource` и `DataField` — для определения требуемого поля набора данных. Для синхронного просмотра следует выбирать такое поле, значения которого не дают пользователю полной информации об объекте и совпадают с ключевым полем в таблице синхронного просмотра. Название этого поля может не совпадать с названием ключевого поля, но типы данных должны быть одинаковыми.

Примечание

При проектировании баз данных желательно, чтобы такие поля все же носили одинаковые названия.

Теперь необходимо задать таблицу синхронного просмотра, ключевое поле и поле синхронного просмотра.

Набор данных, содержащий указанные поля, определяется через соответствующий компонент `TDataSource` в свойстве `ListSource`.

Ключевое поле задается свойством `KeyField`. Во время работы компонента в свойстве `KeyValue` содержится текущее значение, которое связывает между собой два набора данных.

Поле синхронного просмотра определяется свойством `ListField`. Здесь можно задавать сразу несколько полей, которые будут отображаться в компоненте синхронного просмотра. Названия полей разделяются точкой с запятой. Если свойство не определено, то в компоненте будут отображаться значения ключевого поля. Свойство `ListFieldIndex` необходимо для выбора основного поля из списка. Дело в том, что компоненты синхронного просмотра поддерживают механизм наращиваемого поиска, который позволяет быстро находить нужное значение в больших списках. Свойство `ListFieldIndex` определяет, какое поле используется при наращиваемом поиске. В компоненте `TDBlookupComboBox` свойство `ListFieldIndex` также определяет, какое поле будет передано в строку редактирования. Таблица 19.7 содержит основные свойства, включающие механизм синхронного просмотра.

Таблица 19.7. *Основные свойства, включающие механизм синхронного просмотра*

Объявление	Тип	Описание
`property KeyField: string;`	Pb	Ключевое поле таблицы синхронного просмотра
`property KeyValue: Variant;`	Pu	Текущее значение ключевого поля
`property ListField: string;`	Pb	Поле или список полей синхронного просмотра в таблице синхронного просмотра
`property ListFieldIndex: Integer;`	Pb	Номер основного поля синхронного просмотра (используется, когда свойство `ListField` содержит список полей)
`property ListSource: TDataSource;`	Pb	Указывает на компонент `TDataSource`, связанный с таблицей синхронного просмотра
`property NullValueKey: TShortCut;`	Pb	Определяет комбинацию клавиш, нажатие на которые задает нулевое значение поля

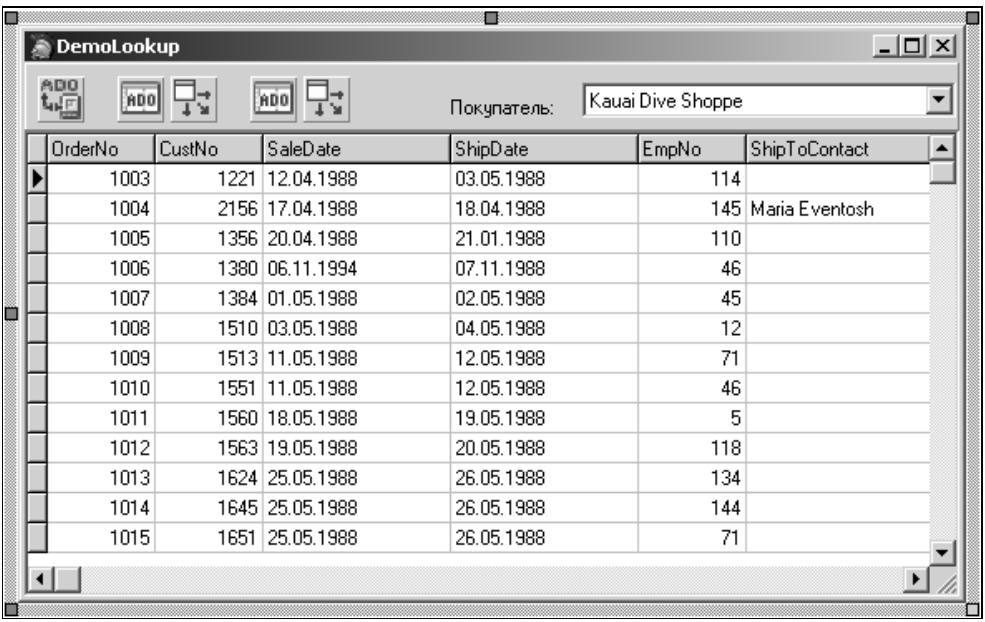


Рис. 19.5. Главная форма проекта DemoLookup

В качестве примера рассмотрим приложение DemoLookup (рис. 19.5), в котором с набором данных таблицы `Orders` из базы данных DBDEMOS связа-

ны компоненты TDBGrid и TDBLookupComboBox. Во втором компоненте при перемещении по записям набора данных отображается имя покупателя, оформившего текущий заказ.

Ключевые свойства компонента настроены следующим образом.

Свойство ListSource указывает на компонент CustSource типа TDataSource, который связан с набором данных синхронного просмотра CustTable.

Свойство ListField указывает на поле Company, все значения которого доступны в списке компонента.

Свойство KeyField указывает на поле CustNo, которое имеет в двух таблицах, и по которому осуществляется связь.

Рассмотрим основные свойства и методы самих компонентов отображения данных, за исключением тех, которые представлены в табл. 19.7 и полностью идентичны для двух компонентов.

Компонент *TDBLookupListBox*

Компонент представляет собой список значений поля синхронного просмотра для поля, заданного свойством DataField, из набора данных DataSource. Его основное назначение — автоматически устанавливать соответствие между полями двух наборов данных по одинаковому значению заданного поля исходной таблицы и ключевого поля таблицы синхронного просмотра. В списке синхронного просмотра отображаются возможные значения для редактирования поля основной таблицы.

По своим функциональным возможностям компонент совпадает с компонентом TDBListBox.

Компонент *TDBLookupComboBox*

Компонент представляет собой комбинированный список значений поля синхронного просмотра для поля, заданного свойством DataField, из набора данных DataSource. Его основное назначение — автоматически устанавливать соответствие между полями двух наборов данных по одинаковому значению заданного поля исходной таблицы и ключевого поля таблицы синхронного просмотра. В списке синхронного просмотра отображаются возможные значения для редактирования поля основной таблицы.

По своим функциональным возможностям компонент совпадает с компонентом TDBComboBox.

Графическое представление данных

Для представления данных из некоторого набора данных в виде графиков различных видов предназначен компонент `TDBChart`. В нем можно одновременно показывать графики для нескольких полей данных. Графики строятся на основе всех имеющихся в наборе данных значений полей. Функционально компонент ничем не отличается от компонента `TChart`.

Настройка параметров компонента осуществляется специальным редактором, который можно открыть двойным щелчком на перенесенном на форму компоненте (рис. 19.6).

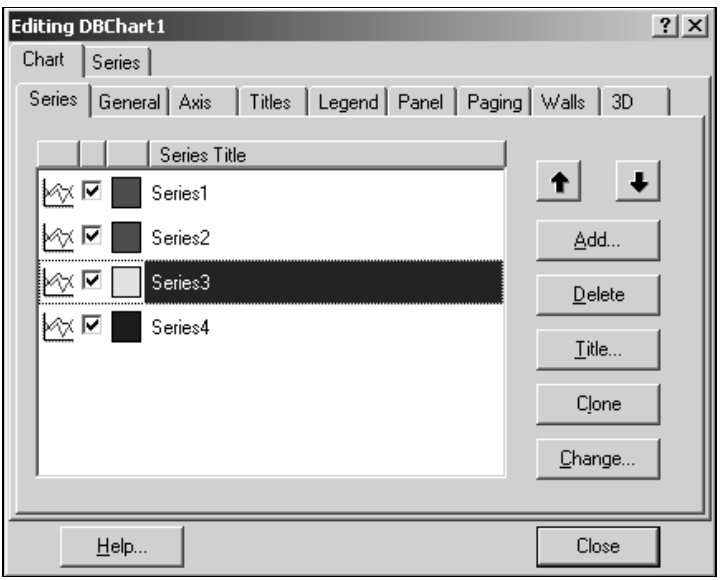


Рис. 19.6. Специализированный редактор компонента `TDBChart`

Здесь мы не будем подробно останавливаться на богатейших изобразительных возможностях этого компонента, рассмотрим только процесс подключения к нему набора данных и построение графиков.

Основой любого графика в компоненте `TDBChart` является так называемая серия, свойства которой представлены классом `TChartSeries`. Для того чтобы построить график значений некоторого поля набора данных, необходимо выполнить следующие действия, большинство из которых выполняется в специализированном редакторе компонента.

1. Создать новую серию и определить ее тип.
2. Задать для серии набор данных.

3. Связать с осями координат нужные поля набора данных и, в зависимости от типа серии, задать дополнительные параметры.
4. Открыть набор данных.

Редактор имеет две главные страницы: **Chart** и **Series**. Страница **Chart** содержит многостраничный блокнот и предназначена для настройки параметров самого графика. Страница **Series** также содержит многостраничный блокнот и используется для настройки серий значений данных.

Для создания новой серии необходимо в редакторе перейти на главную страницу **Chart**, а на ней открыть страницу **Series** (см. рис. 19.6). На этой странице нужно щелкнуть на кнопке **Add**, а затем в появившемся диалоге выбрать тип серии. После этого в списке на странице **Series** появляется строка новой серии. Здесь можно переопределить тип, цвет и видимость серии, щелкнув на соответствующей зоне строки.

Все остальные страницы блокнота на главной странице **Chart** предназначены для настройки параметров графика.

Теперь необходимо перейти на главную страницу **Series** и на ней из списка названий серий выбрать необходимую. После этого на странице **Data Source** из списка выбирается строка **DataSet**. После этого в появившемся списке **DataSet** выбирается нужный набор данных.

Список **X** позволяет выбрать поле набора данных, значения которого будут последовательно откладываться по оси абсцисс. Список **Y** позволяет выбрать поле набора данных, значения которого будут отложены по оси ординат. Соответствие между значениями полей по двум осям определяется принадлежностью к одной записи набора данных. Выбор поля в списке **Labels** привязывает его значения в виде меток к оси абсцисс.

Примечание

Здесь описан набор элементов управления для линейного типа серии. Для других типов элементы управления могут отличаться.

Теперь осталось только открыть набор данных, и компонент `TDBChart` построит график.

Аналогичным образом на этот же компонент можно поместить и другие графики.

Свойства и методы компонента `TDBChart` представлены в табл. 19.8.

Таблица 19.8. *Свойства и методы компонента* `TDBChart`

Объявление	Описание
Свойства	
`property AutoRefresh : Boolean;`	Разрешает или запрещает обновление данных в серии при открытии связанного набора данных
`property RefreshInterval : LongInt;`	Задает временной интервал в секундах между обновлениями данных в сериях из связанных наборов данных
`property ShowGlassCursor : Boolean;`	Разрешает показ курсора "песочные часы" при обновлениях данных
Методы	
`procedure CheckDataSource;`	Обновляет данные в сериях
`function IsValidDataSource(ASeries: TChartSeries; AComponent: TComponent): Boolean; virtual;`	Проверяет, связан ли набор данных `AComponent` с серией `ASeries`. В случае успеха проверки возвращает `True`
`procedure RefreshData;`	Обновляет данные во всех сериях
`procedure RefreshDataSet(ADataSet: TDataSet; ASeries: TChartSeries);`	Считывает все записи в наборе данных `AdataSet` и переносит их в серию `ASeries`
Методы-обработчики событий	
`property OnProcessRecord : TProcessRecordEvent;`	Вызывается при переносе данных из отдельной записи набора данных в серию

Резюме

Компоненты отображения играют важную роль при создании интерфейсов приложений баз данных. Разнообразие предлагаемых элементов управления позволяет решать любые задачи по организации взаимодействия пользователя с базой данных. Все они взаимодействуют с набором данных через компонент `TDataSource`.

ЧАСТЬ V

Технологии доступа к данным

Глава 20

Процессор баз данных Borland Database Engine

Любое приложение баз данных имеет в своем составе или использует сторонний механизм доступа к данным, который берет на себя подавляющее большинство стандартных низкоуровневых операций работы с базами данных. Например, любое такое приложение при открытии таблицы БД должно выполнить примерно одинаковый набор операций:

- найти местоположение базы данных;
- найти таблицу, открыть ее и прочитать служебную информацию;
- прочитать данные в соответствии с форматом хранения данных и т. д.

Очевидно, что если все стандартные функции доступа к данным реализовать в виде специальной программы, сервиса или динамической библиотеки, то это существенно упростит разработку приложений баз данных, которым для выполнения той или иной операции достаточно будет вызвать готовую процедуру.

Одним из традиционных способов взаимодействия приложения, созданного в среде разработки Delphi и базы данных, является использование процессора баз данных Borland Database Engine. Он представляет собой набор динамических библиотек, функции которых позволяют не только обращаться к данным, но и эффективно управлять ими на стороне приложения.

Для работы с источниками данных при посредстве BDE в Delphi имеется специальный набор компонентов, расположенных в категории **BDE** Палитры инструментов. Эти компоненты работают с базами данных с помощью возможностей BDE, обращаясь к его функциям и процедурам. Механизм доступа к BDE инкапсулирован в базовом классе `TBDEDataSet`. Поэтому в процессе программирования у вас не будет необходимости вызывать функции BDE напрямую. Почти все, что можно сделать путем прямого обращения, можно сделать и через компоненты, а это проще и надежнее.

Тем не менее внутреннюю организацию механизма доступа к данным всегда полезно знать. Кроме этого, всегда полезно знать и уметь приме-

нять дополнительные возможности, которые BDE может предоставить разработчику.

BDE взаимодействует с базами данных при посредстве драйверов. Для наиболее распространенных локальных СУБД разработан набор стандартных драйверов. Работа с наиболее распространенными серверами БД осуществляется при помощи драйверов системы SQL Links. Кроме этого, если для базы данных существует драйвер ODBC, то можно применить и его. Достаточно зарегистрировать этот драйвер в BDE.

Однако BDE не претендует на всеобъемлющую универсальность и имеет некоторые недостатки. Это, например, снижение скорости работы приложения, недостатки реализации некоторых драйверов и т. д. Сейчас технология доступа к данным BDE считается устаревшей и Borland рекомендует вместо нее технологию dbExpress, которая также рассматривается в этой книге.

В этой главе обсуждаются следующие вопросы.

- Архитектура и составные части BDE.
- Псевдонимы БД и настройка драйверов BDE.
- Утилита BDE Administrator.
- Способы прямого использования функций API BDE.
- Компоненты доступа к данным BDE.

Архитектура и функции BDE

BDE представляет собой набор динамических библиотек, которые "умеют" передавать запросы на получение или модификацию данных из приложения в нужную базу данных и возвращать результат обработки. Библиотеки работают с вспомогательными файлами языковой поддержки и информацией о настройках среды.

В составе BDE поставляются стандартные драйверы, обеспечивающие доступ к СУБД Paradox, dBASE, FoxPro и текстовым файлам. Локальные драйверы (рис. 20.1) устанавливаются автоматически совместно с ядром процессора. Один из них можно выбрать в качестве стандартного драйвера, который имеет дополнительные настройки, влияющие на функционирование процессора БД.

Доступ к данным серверов SQL обеспечивает отдельная система драйверов — SQL Links. С их помощью в Delphi можно разрабатывать приложения для серверов Oracle, Informix, Sybase, DB2 и, естественно, InterBase. Эти драйверы необходимо устанавливать дополнительно.

Помимо этого. в BDE имеется очень простой механизм подключения любых драйверов ODBC (к примеру, Microsoft Access) и создания на их основе сокетов ODBC.

Примечание

С точки зрения пользователя процесс подключения локального драйвера и драйвера SQL Links практически не отличается, за исключением деталей настройки. Настройка драйверов и собственных параметров BDE осуществляется при помощи специальной утилиты — BDE Administrator и рассматривается далее в этой главе.

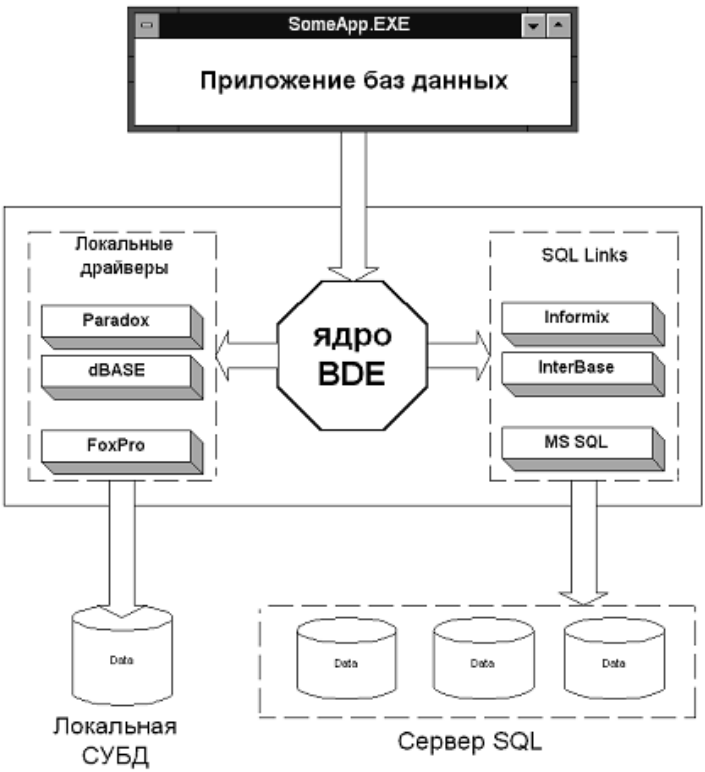


Рис. 20.1. Структура процессора баз данных BDE

В состав BDE входят следующие функциональные подсистемы.

- Администратор системных ресурсов — управляет процессом подключения к данным — при необходимости устанавливает нужные драйверы, а при завершении работы автоматически освобождает занятые ресурсы. Поэтому BDE всегда задействует ровно столько ресурсов, сколько необходимо.
- Система обработки запросов — обеспечивает выполнение запросов SQL или QBE от приложения к любым базам данных, для которых установлен

драйвер, даже если сама СУБД не поддерживает прямое использование запросов SQL.

- Система сортировки — запатентованная технология, обеспечивающая очень быстрый поиск по запросам SQL и через стандартные драйверы для Paradox и dBASE.
- Система пакетной обработки — механизм преобразования данных из одного формата в другой при выполнении операций над целыми таблицами. Эта система является основой для компонента `TBatchMove` и утилиты DataPump (автоматического переноса структур данных между базами данных), входящей в стандартную поставку BDE.
- Менеджер буфера — управляет единой для всех драйверов буферной областью памяти, с которой одновременно могут взаимодействовать несколько драйверов. Это позволяет существенно экономить системные ресурсы.
- Менеджер памяти — взаимодействует с ОС и обеспечивает эффективное использование выделяемой памяти. Ускоряет работу драйверов, которые для получения небольших фрагментов памяти обращаются к нему, а не к ОС. Дело в том, что менеджер памяти выделяет большие объемы оперативной памяти и затем распределяет ее небольшими кусками между драйверами согласно их потребностям.
- Транслятор данных — обеспечивает преобразование форматов данных для различных типов БД.
- Кэш BLOB — ускоряет работу с данными в формате BLOB.
- SQL-генератор — транслирует запросы в формате QBE в запросы SQL.
- Система реструктуризации — преобразует наборы данных в таблицы Paradox или dBASE.
- Система поддержки драйверов SQL — повышает эффективность механизма поиска при выполнении запросов SQL.
- Таблицы в памяти — позволяют создавать таблицы непосредственно в оперативной памяти. Используется для ускорения обработки больших массивов данных, сортировки, преобразования форматов данных.
- Связанные курсоры — обеспечивают низкоуровневое выполнение межтабличных соединений. Позволяют разработчику не задумываться над реализацией подобных связей при работе на уровне VCL — для этого достаточно установить значения нескольких свойств.
- Менеджер конфигурации — обеспечивает разработчику доступ к информации о конфигурации драйверов.

Перечисленные функции реализованы в динамических библиотеках, которые, собственно, и называются процессором БД (табл. 20.1).

Таблица 20.1. *Ядро процессора баз данных BDE 5*

Имя файла	Назначение
IDAPI32.DLL	Базовая динамическая библиотека BDE
IDPROV.DLL	Динамическая библиотека, отвечающая за работу серверной части приложения
BLW32.DLL	Динамическая библиотека, обеспечивающая поддержку драйверов национальных языков
IDBAT32.DLL	Динамическая библиотека с функциями межтабличного переноса данных
IDQBE32.DLL	Динамическая библиотека, обеспечивающая работу *запросов по примеру* (Query By Example, QBE)
IDSQL32.DLL	Динамическая библиотека, обеспечивающая обработку запросов SQL
IDASCI32.DLL	Динамическая библиотека, обеспечивающая работу драйвера текстовых файлов
IDPDX32.DLL	Динамическая библиотека, обеспечивающая работу драйвера Paradox
IDDBAS32.DLL	Динамическая библиотека, обеспечивающая работу драйвера dBASE
DODBC32.DLL	Динамическая библиотека, обеспечивающая работу драйвера сокета ODBC
IDR20009.DLL	Динамическая библиотека ресурсов, содержащая сообщения об ошибках
IDDAO32.DLL	Динамическая библиотека, обеспечивающая работу драйвера Microsoft Access 95 и Jet Engine 3.0
IDDA3532.DLL	Динамическая библиотека, обеспечивающая работу драйвера Microsoft Access 97 и Jet Engine 3.5
IDDR32.DLL	Динамическая библиотека для работы с репозиторием данных

Кроме этого, имеется шесть дополнительных файлов DLL, обеспечивающих работу BDE с серверами Oracle и Microsoft SQL Server.

Псевдонимы баз данных и настройка BDE

Для успешного доступа к данным приложение и BDE должны обладать информацией о местоположении файлов требуемой базы данных. Задание маршрута входит в обязанности разработчика.

Самый простой способ заключается в явном задании полного пути к каталогу, в котором хранятся файлы БД. Но при изменении пути, что случается не так уж редко, например, при переносе готового приложения на компьютер заказчика, разработчик должен перекомпилировать проект с учетом будущего местонахождения БД или предусмотреть специальные элементы управления, в которых можно задать путь к БД.

Для решения такого рода проблем разработчик может использовать псевдоним базы данных, который представляет собой именованную структуру, содержащую путь к файлам БД и некоторые дополнительные параметры. В первом приближении можно сказать, что вы просто присваиваете маршруту произвольное имя, которое используется в приложении. Тогда при переносе приложения на компьютере заказчика достаточно создать стандартными средствами BDE одноименный псевдоним и настроить его на нужный каталог. При этом само приложение не требует переделок, т. к. оно обращается к псевдониму с одним именем, а вот BDE уже "знает", куда отправить запрос приложения, использовавшего этот псевдоним.

Помимо маршрута к файлам базы данных псевдоним BDE обязательно содержит информацию о драйвере доступа к данным БД. Наличие других параметров зависит от типа драйвера, а значит, от типа СУБД.

Для управления псевдонимами баз данных, настройки стандартных и дополнительных драйверов в составе BDE имеется специальная утилита: BDE Administrator (исполняемый файл BDEADMIN.EXE).

Стандартная конфигурация BDE сохраняется в файле IDAPI.CFG. При необходимости текущую конфигурацию можно сохранить в новом файле с расширением cfg или загрузить заново при помощи команд **Save As Configuration** и **Open Configuration** из меню **Object**.

В верхней части окна утилиты расположена панель инструментов, кнопки которой необходимы при работе с конкретным элементом настройки BDE.

Рабочая область утилиты BDE Administrator представляет собой двухстраничный блокнот (рис. 20.2).

Страница **Databases** содержит иерархическое дерево, в узлах которого расположены установленные в системе на данный момент псевдонимы БД. При выборе какого-либо псевдонима в правой части панели появляются путь к файлам базы данных и перечень параметров драйвера, соответствующего псевдониму, которые можно настраивать вручную.

Страница **Configuration** используется для настройки параметров драйверов BDE, предназначенных для обеспечения доступа к локальным СУБД и серверам БД. Также здесь определяется системная конфигурация BDE, которая включает параметры числовых форматов, дат, времени. Вся информация на этой странице также структурирована в виде иерархического дерева (рис. 20.3).

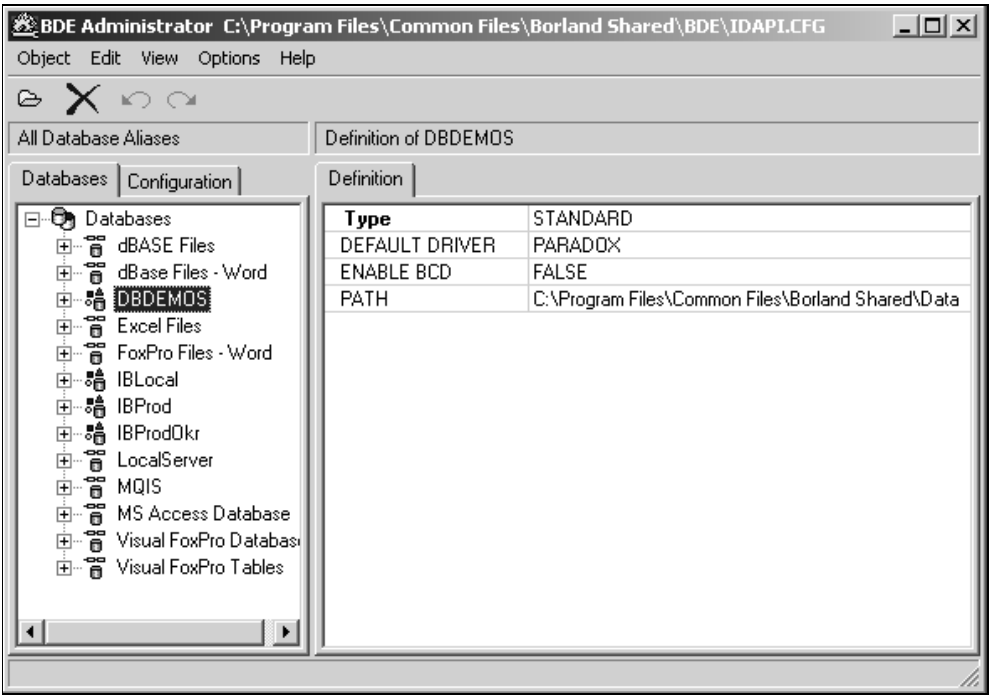


Рис. 20.2. Окно утилиты BDE Administrator с открытой страницей **Databases**

При выборе в левой панели утилиты какого-либо узла, в правой части на странице **Definition** отображается вся необходимая информация для этого объекта.

Сохранение изменений осуществляется при помощи команд меню **Object**, всплывающего меню или при перемещении на другой псевдоним.

Для создания нового псевдонима требуется выбрать команду **New** из меню **Object** или из всплывающего меню узла **Databases** на одноименной странице. Затем в появившемся простом диалоге задается необходимый драйвер.

Отметим, что один из четырех стандартных локальных драйверов устанавливается на странице **Configuration** в качестве предопределенного, поэтому в списке он доступен под названием STANDARD, а остальные не видны вообще. Из драйверов SQL Links доступны те, которые были установлены при инсталляции Delphi или позже.

Кроме того, в списке можно выбрать один из драйверов ODBC, установка которых осуществляется стандартными системными средствами в Панели управления Windows.

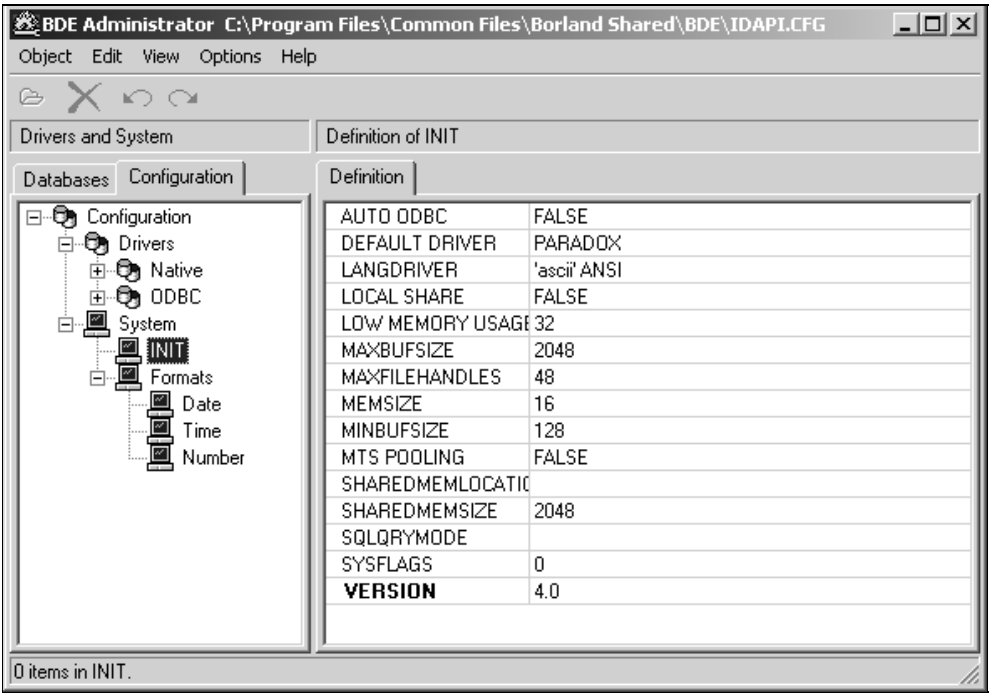


Рис. 20.3. Окно утилиты BDE Administrator с открытой страницей **Configuration**

После выбора драйвера в дереве псевдонимов БД появляется новый узел, для драйвера которого требуется установить необходимые параметры.

Для четырех локальных драйверов список параметров в правой части панели утилиты на странице **Definition** ограничивается параметрами стандартного драйвера (STANDARD), подробная настройка для каждого драйвера осуществляется на странице **Configuration**.

Назначение параметров локальных драйверов BDE (Paradox, dBASE, FoxPro, ASCII) представлено в табл. 20.2.

Таблица 20.2. *Параметры драйверов BDE для локальных баз данных*

Параметр	Назначение
STANDARD	
DEFAULT DRIVER	Задает тип конкретного локального драйвера (Paradox, dBASE, FoxPro, ASCIIDRV)

Таблица 20.2 *(продолжение)*

Параметр	Назначение
STANDARD	
ENABLE BCD	Определяет способ представления вещественных чисел. При значении `True` такие числа преобразуются в формат BCD (Binary Coded Decimals — десятичные с двоичным кодированием). Точность составляет 20 знаков после запятой
PATH	Указывает путь к файлам базы данных
PARADOX	
NET DIR	Указывает путь к файлу обеспечения сетевого доступа к БД PDOXUSRS.NET. Драйвер приложения, которое работает с БД локально, должен указывать на этот файл, расположенный на том же компьютере. Драйвер приложения, обращающегося к БД по сети, должен указывать на подключенный сетевой диск с этим файлом
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Для Paradox имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки (используйте драйвер Paradox Cyrr 866)
BLOCK SIZE	Задает размер блоков дискового пространства для хранения записей, кратный 1024
FILL FACTOR	Определяет процент заполнения блока дискового пространства при хранении индексов, по умолчанию 95%
LEVEL	Задает формат временной таблицы в памяти: 3 — совместим с Paradox 3.5 и ниже. 4 — Paradox 4.0. 5 — Paradox 5.0. 7 — Paradox для Win32
STRICTINTEGRTY	Определяет возможность использования приложениями на базе Paradox 4.0 более поздних таблиц со ссылочной целостностью. При значении `True` использование разрешается, но возникает риск нарушения целостности данных
dBASE	
VERSION	Нередактируемая информация о версии драйвера

Таблица 20.2 *(окончание)*

Параметр	Назначение
dBASE	
TYPE	Тип СУБД. Для dBASE имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки (используйте драйвер dBASE RUS cp866)
LEVEL	Задает формат таблиц. Значение соответствует номеру версии СУБД
MDX BLOCK SIZE	Размер блоков для файлов с расширением mdx, кратный 512
MEMO FILE BLOCK SIZE	Размер блоков для файлов с данными типа `Memo` (расширение dbt), кратный 512
FOXPRO	
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Для FoxPro имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки
LEVEL	Имеет значение 25

Примечание

Драйвер текстовых файлов ASCIIDRV имеет параметры стандартного драйвера.

Назначение параметров драйверов SQL Links для серверов SQL представлено в табл. 20.3. Сначала приведены параметры, которые встречаются в двух и более драйверах, затем уникальные для каждого драйвера параметры. Драйверы для серверов InterBase и Sybase не представлены, т. к. содержат только общие для двух серверов параметры.

Таблица 20.3. *Параметры драйверов BDE для серверов SQL*

Параметр	Назначение
Общие параметры (встречаются как минимум у двух драйверов)	
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Только для чтения

Таблица 20.3 *(продолжение)*

Параметр	Назначение
Общие параметры (встречаются как минимум у двух драйверов)	
DLL	Название библиотеки динамического связывания SQL Links для 16-разрядного драйвера. Только для чтения
DLL32	Название библиотеки динамического связывания SQL Links для 32-разрядного драйвера. Только для чтения
DRIVER FLAGS	Используется только при необходимости применения старых версий драйвера, где не поддерживается уровень изоляции транзакций Read Committed. Для этого необходимо установить значение 512
TRACE MODE	Содержит битовую маску, которая определяет тип выдаваемой отладочной информации
BATCH COUNT	Задает число записей, модифицируемых в одном пакете при фиксации транзакций
BLOB SIZE	Размер кэша для данных типа BLOB. Диапазон от 32K до 1000K
BLOBS TO CACHE	Задает число кэшируемых записей с данными BLOB. Диапазон от 64 до 65 536
ENABLE BCD	Определяет способ представления вещественных чисел. При значении `True` такие числа преобразуются в формат BCD (Binary Coded Decimals — десятичные с двоичным кодированием), который позволяет округлять погрешности высших разрядов дробной части числа. Изменение параметра для псевдонима работает только, если параметр драйвера на странице **Configuration** непустой
ENABLE SCHEMA CACHE	Определяет режим кэширования структуры данных. При значении `True` структура таблиц БД кэшируется локально в каталоге, задаваемом параметром SCHEMA CACHE DIR. Рекомендуется использовать только для баз данных с постоянной структурой
LANGDRIVER	Определяет драйвер языковой поддержки
MAX ROWS	Ограничивает максимальное число записей, которое может быть передано клиенту в ответ на запрос. Значение по умолчанию –1 (ограничений нет)
OPEN MODE	Режим работы с записями БД: READ/WRITE — полный доступ. READ ONLY — только чтение

Таблица 20.3 *(продолжение)*

Параметр	Назначение
Общие параметры (встречаются как минимум у двух драйверов)	
SCHEMA CACHE DIR	Каталог для локального кэширования структуры данных (см. параметр ENABLE SCHEMA CACHE)
SCHEMA CACHE SIZE	Задает число таблиц, структура данных которых может кэшироваться
SCHEMA CACHE TIME	Задает время хранения кэшируемой структуры данных: –1 — время не ограничено. 0 — данные не кэшируются. 1 — 2147483647 сек
SERVER NAME	Указывает путь к таблицам БД (это может быть локальный маршрут или маршрут с указанием удаленного сервера БД)
SQLPASSTHRU MODE	Задает способ разделения соединения с сервером между прямыми запросами SQL и запросами, управляемыми пользователем: SHARED AUTOCOMMIT — соединение используется совместно и прямые запросы фиксируются автоматически. SHARED NO AUTOCOMMIT– соединение используется совместно и прямые запросы фиксируются сервером самостоятельно. NOT SHARED — совместное использование запрещено
SQLQRYMODE	Задает режим управления запросами: NULL — сначала запрос передается серверу, и если тот не может обработать его, запрос выполняется локально. SERVER — запрос передается серверу. LOCAL — запрос выполняется локально
VENDOR INIT	Название файла динамической библиотеки поставщика
CONNECT TIMEOUT	Определяет временной интервал, после которого клиент попытается восстановить прерванную связь с сервером
TIMEOUT	Задает время ожидания ответа сервера на запрос
BLOB EDIT LOGGING	Управляет механизмом сохранения всех изменений для полей типа BLOB. При значении `True` изменения сохраняются
DATABASE NAME	Имя базы данных

Таблица 20.3 *(продолжение)*

Параметр	Назначение
Общие параметры (встречаются как минимум у двух драйверов)	
MAX QUERY TIME	Задает максимальное время ожидания ответа на запрос
USER NAME	Имя пользователя, которое используется сервером при подключении
Microsoft SQL Server (MSSQL)	
MAX DBPROCESSES	Максимальное число процессов, одновременно работающих в данном соединении
APPLICATION NAME	Имя приложения, помогающее серверу идентифицировать процессы
DATE MODE	Определяет формат даты: 0 — МДГ. 1 — ДМГ. 2 — ГМД
HOST NAME	Содержит имя рабочей станции. Помогает серверу при идентификации процессов
NATIONAL LANG NAME	Задает национальный язык, который используется для вывода текста в сообщениях об ошибках
TDS PACKET SIZE	Определяет размер пакетов потоков данных
Oracle (ORACLE)	
NET PROTOCOL	Устанавливает сетевой протокол передачи данных
Informix (INFORMIX)	
DATE SEPARATOR	Задает разделитель для формата даты
Microsoft Access (MSACCESS)	
SYSTEM DATABASE	Путь к системной базе данных с информацией о правах доступа. При изменении параметра драйвер необходимо перезагрузить
DB2 (DB2)	
DB2 DSN	Задает имя соединения с БД. Это название псевдонима клиента DB2, который создается на сервере
DRIVER	Имя драйвера DB2
ROWSET SIZE	Определяет число записей, передаваемых одновременно

Таблица 20.3 *(окончание)*

Параметр	Назначение
Драйверы ODBC	
ODBC DRIVER	Имя драйвера ODBC
ODBC DSN	Имя набора данных ODBC

После настройки параметров драйвера и сохранения текущей конфигурации новый псевдоним становится доступен для любого приложения, использующего BDE.

Страница **Configuration**, помимо настройки установленных в BDE драйверов, позволяет редактировать параметры, устанавливаемые BDE при инициализации приложения. Эти параметры доступны при выборе узлов **System**, а затем **INIT** иерархического дерева. Назначение параметров представлено в табл. 20.4.

Таблица 20.4. *Параметры инициализации приложения*

Параметр	Назначение
AUTO ODBC	В значении `True` при каждой инициализации в BDE автоматически импортируются все установленные в системе драйверы ODBC
DATA REPOSITORY	Имя текущего словаря данных
DEFAULT DRIVER	Локальный драйвер, используемый по умолчанию в драйвере STANDARD
LANGDRIVER	Драйвер языковой поддержки. При использовании стандартных локальных драйверов это значение перекрывает те, которые определены непосредственно в конфигурациях драйверов Paradox, dBASE, FoxPro, ASCII
LOCAL SHARE	Устанавливает режим совместного использования файлов приложениями, работающими через BDE, и другими программами. При значении `True` совместное использование разрешено
LOW MEMORY USAGE LIMIT	Максимальный объем памяти (в килобайтах), который BDE пытается использовать в первом мегабайте оперативной памяти
MAXBUFSIZE	Максимальный размер кэша данных. Он должен быть не меньше значения параметра MINBUFSIZE и кратен 128
MAXFILEHANDLES	Максимальное число используемых файлов

Таблица 20.4 *(окончание)*

Параметр	Назначение
MEMSIZE	Максимальный объем используемой BDE памяти в мегабайтах
MINBUFSIZE	Минимальный размер кэша данных. Он должен быть не больше значения параметра MAXBUFSIZE и кратен 128
MTS POOLING	Управляет режимом объединения ресурсов MTS (Microsoft Transaction Server). Обеспечивает лучшую производительность
SHAREDMEMLOCATION	Содержит адрес памяти, который пытаются использовать Менеджер памяти и Менеджер буфера. При возникновении конфликтов адрес необходимо поменять вручную. Задается только второе слово адреса
SHAREDMEMSIZE	Максимальный объем памяти, используемый Менеджером памяти и Менеджером буфера
SQLQRYMODE	См. табл. 20.2
SYSFLAGS	Не используется
VERSION	Номер внутренней версии BDE. Только для чтения

Также на странице **Configuration** устанавливаются параметры форматов даты, времени и чисел. Доступ к параметрам осуществляется через узлы **System** и **Format**.

Интерфейс прикладного программирования BDE

Как уже говорилось, любое приложение Delphi, работающее с базами данных и написанное на основе стандартных компонентов доступа к данным, обращается к данным и получает результат при помощи BDE. При этом механизм доступа к данным использует вызовы функций из API BDE.

Достаточно сложно представить себе такую ситуацию, когда возникает необходимость создания приложения, задействующего только функции BDE, без применения компонентов доступа к данным VCL. А вот отдельные функции вполне могут понадобиться в любой программе. Поэтому рассмотрим процесс работы приложения, содержащего вызовы BDE, т. к. это дает хорошую возможность понять механизм доступа к данным, который реализован в Delphi.

Итак, для создания приложения на основе вызовов функций BDE необходимо выполнить следующие операции.

1. Инициализация BDE (функция `DbiInit`).
2. Открытие объекта базы данных (функция `DbiOpenDatabase`).
3. Определение рабочего каталога (функция `DbiSetDirectory`), если на предыдущем этапе не задается псевдоним БД.
4. Определение временного каталога (функция `DbiSetPrivateDir`).
5. Открытие набора данных и создание курсора (функции `DbiOpenTable`, `DbiQExec` и др.; дескриптор курсора `hDBICur`).
6. Заполнение структуры `CURProps`, содержащей данные о курсоре и наборе данных (функция `DbiGetCursorProps`).
7. Выделение памяти для буфера записи.
8. Навигация по набору данных (функции `DbiSetToBegin`, `DbiSetToEnd`, `DbiSetToCursor` и др.)
9. Чтение необходимой записи (функции `DbiGetRelativeRecord`, `DbiGetNextRecord`, `DbiGetRecord`, `DbiGetPriorRecord` и др.).
10. Чтение или обновление необходимого поля (функции `DbiGetField`, `DbiPutField`).
11. Освобождение всех ресурсов (освобождение буфера записи, закрытие курсора, таблицы, BDE).

При использовании в программе функций из API BDE необходимо включить в секцию `uses` модуль BDE.

В прикладном программировании задачи, которые требовали бы выполнения всех описанных операций, практически не встречаются. Между тем включение в программный код отдельных функций API BDE оправдано. В качестве примера рассмотрим приложение, которое позволяет очистить таблицу базы данных (рис. 20.4).

Дело в том, что при удалении записи из таблицы локальной СУБД (например, Paradox) размер файла таблицы остается прежним, даже если удалить все записи. Следовательно, на самом деле запись не удаляется, а только становится недоступной. При интенсивной работе с базой данных файлы таблиц могут занимать значительные объемы дискового пространства при довольно умеренном числе записей.

Полная очистка таблиц базы данных осуществляется функцией `DbiEmptyTable` из API BDE. Именно она используется в демонстрационном приложении для радикального уменьшения размера таблиц.

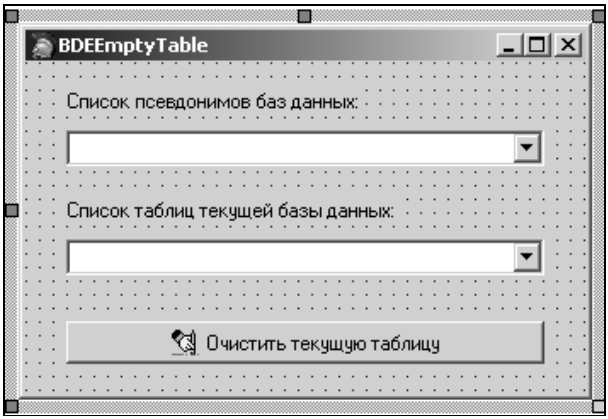


Рис. 20.4. Главная форма проекта BDEEmptyTable

Примечание

Функция `DbiEmptyTable` используется в методе `EmptyTable` компонентов доступа к данным (см. следующую главу).

В листинге 20.1 приведен исходный код этого приложения. Помимо указанной функции в нем содержатся функции создания списка параметров доступных баз данных и таблиц текущей базы данных.

Листинг 20.1. Модуль главной формы приложения BDEEmptyTable

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, BDE, ExtCtrls, DBCtrls, Grids, DBGrids, Db, DBTables,
Buttons;

type
  TMainForm = class(TForm)
    AliasesList: TComboBox;
    TablesList: TComboBox;
    EmptyBtn: TBitBtn;
    Label1: TLabel;
    Label2: TLabel;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
```

```
    procedure FormShow(Sender: TObject);
    procedure AliasesListChange(Sender: TObject);
    procedure EmptyBtnClick(Sender: TObject);
  private
    hDB: hDBIDB;
    hCursor: hDBICur;
    DBDesc: DBDesc;
    TblDesc : TBLBaseDesc;
  public
    { Public declarations }
  end;
var
  MainForm: TMainForm;
implementation
{$R *.DFM}
procedure TMainForm.FormShow(Sender: TObject);
var Rslt: DBIResult;
begin
 AliasesList.Items.Clear;
 TablesList.Items.Clear;
 hDB := Nil;
 try
  DbiInit(Nil);   // Инициализация BDE
  DbiOpenDatabaseList(hCursor);
  repeat
   Rslt:= DbiGetNextRecord(hCursor, dbiNOLOCK, @DBDesc, nil);
   if (Rslt <> DBIERR_EOF) then
   AliasesList.Items.Add(StrPas(DBDesc.szName));
  until (rslt <> DBIERR_NONE);
  DbiCloseCursor(hCursor);
 except
  on E:EDBEngineError do ShowMessage('Ошибка инициализации BDE');
 end;
end;
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 try
 finally
  if hDB <> Nil then DbiCloseDatabase(hDB);// Закрытие базы данных
```

```
  DbiExit;                                  // Закрытие сеанса работы с BDE
 end
end;
procedure TMainForm.AliasesListChange(Sender: TObject);
begin
 try
  if hDB <> Nil
  then DbiCloseDatabase(hDB);// Закрытие базы данных
  DbiOpenDatabase            // Открытие базы данных
  (
   PChar(AliasesList.Text),  // Псевдоним базы данных
   Nil,                      // Тип базы данных
   dbiReadWrite,             // Режим редактирования данных
   dbiOpenShared,            // Режим разделения данных
   Nil,                      // Пароль
   0,                        // Число дополнительных параметров
   Nil,                      // Перечень полей для доп. параметров
   Nil,                      // Список доп. параметров
   hDB                       // Дескриптор базы данных
  );
  DbiSetPrivateDir('c:\temp');// Определение временного каталога
  DbiOpenTableList(hDb, False, False, '*.DB', hCursor);
  TablesList.Items.Clear;
  TablesList.Clear;
  while DbiGetNextRecord(hCursor, dbiNOLOCK, @TblDesc, nil) = dbiErr_None
   do TablesList.Items.Add(TblDesc.szName);
  DbiCloseCursor(hCursor);
 except
  on E:EDBEngineError do ShowMessage('Ошибка открытия базы данных');
 end;
end;
procedure TMainForm.EmptyBtnClick(Sender: TObject);
begin
 try
 DbiEmptyTable(hDB, Nil, PChar(TablesList.Text), '');
 except
  on E:EDBEngineError do ShowMessage('Неверно задана таблица');
 end;
end;
end.
```

При открытии главной формы (метод-обработчик `FormShow`) функция `DbiInit` осуществляет инициализацию BDE. Затем функция `DbiOpenDatabaseList` создает в памяти временную таблицу, в которую записываются характеристики каждой зарегистрированной базы данных. Для этого применяется структура `DBDesc`. Курсор `hCursor` обеспечивает доступ к записям о базах данных.

После этого функция `DbiGetNextRecord` позволяет осуществить последовательное считывание имен псевдонимов баз данных (для этого в параметре передается указатель на структуру `DBDesc`) и их запись в список компонента `AliasesList` типа `TComboBox`.

При выборе из этого списка конкретного псевдонима работает метод-обработчик `AliasesListChange`. В нем открывается соответствующая база данных (функция `DbiOpenDatabase`), доступ к которой в дальнейшем осуществляется через дескриптор `hDB`.

Примечание

Открытие базы данных в примере настроено в "минимальной конфигурации" для демонстрационной базы данных DBDEMOS. Для того чтобы пример мог открывать любые базы данных, необходимо правильно передавать все параметры в функцию `DbiOpenDatabase`.

Функция `DbiOpenTableList` создает временную таблицу в памяти, в которую помещаются данные о таблицах выбранной базы данных в соответствии с форматом структуры `TBLBaseDesc`, а `DbiGetNextRecord` позволяет передать эту информацию в список компонента `TablesList` типа `TComboBox`.

При щелчке на кнопке `EmptyBtn` в методе-обработчике `EmptyBtnClick` работает функция `DbiEmptyTable`, которая очищает выбранную ранее в компоненте `TablesList` таблицу.

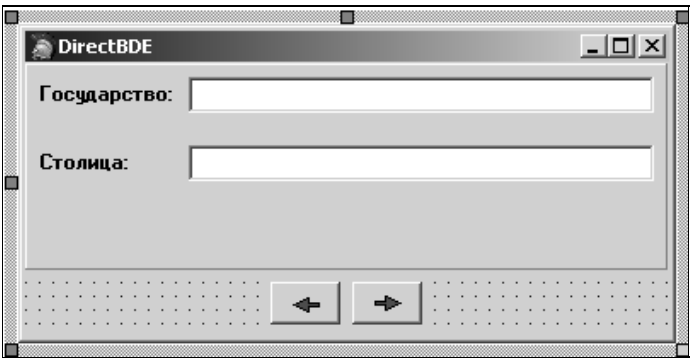


Рис. 20.5. Главная форма проекта DirectBDE

Теперь рассмотрим пример простейшего приложения, которое может отображать два поля из таблицы COUNTRY.DB в демонстрационной базе данных DBDEMOS. Эта база данных поставляется в комплекте Delphi. В примере содержатся только функции API BDE.

Проект называется DirectBDE и имеет только одну форму, в которой отображаются сведения из таблицы COUNTRY.DB о государствах и их столицах (рис. 20.5). Кнопки в нижней части формы позволяют перемещаться по набору данных. Листинг 20.2 иллюстрирует модуль главной формы приложения DirectBDE.

Листинг 20.2. Модуль главной формы приложения DirectBDE

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, Buttons, BDE, ExtCtrls;
type
  TMainForm = class(TForm)
    PriorBtn: TBitBtn;
    Panel2: TPanel;
    NextBtn: TBitBtn;
    Label3: TLabel;
    CountryEdit: TEdit;
    Label1: TLabel;
    CapitalEdit: TEdit;
    procedure PriorBtnClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure NextBtnClick(Sender: TObject);
  private
    hDB: hDBIDB;
    hCur: hDBICur;
    CursProps: CurProps;
    RecBuf: pByte;
    FValue: array[0..255] of Char;
    IsEmpty: Bool;
    procedure OnBDEError;
```

```
  public
  end;
var
  MainForm: TMainForm;

implementation

{$R *.DFM}
procedure TMainForm.OnBDEError;
var ErrInfo: dbiErrInfo; //Структура, содержащая информацию об ошибках
    AStr: String;
begin
 DbiGetErrorInfo(True, ErrInfo);//Функция возвращает информацию об ошибке
 case ErrInfo.iError of
   9733: AStr := 'Для создания записи недостаточно параметров';
  10024: AStr := 'Ошибка доступа к данным';
  10245: AStr := 'База данных занята другим пользователем';
  10038: AStr := 'Значение поля задано неверно';
  11871: AStr := 'Несоответствие типов';
  11959: AStr := 'В выражении отсутствует оператор GROUP BY';
 else
  AStr := 'Ошибочная операция с данными';
 end;
 ShowMessage(AStr);
end;
procedure TMainForm.FormShow(Sender: TObject);
begin
 hDB := Nil;
 hCur := Nil;
 DbiInit(Nil);          // Инициализация системы BDE
 DbiOpenDatabase        // Открытие базы данных
 (
  'DBDEMOS',            // Псевдоним базы данных
  Nil,                  // Тип базы данных
  dbiReadWrite,         // Режим редактирования данных
  dbiOpenShared,        // Режим разделения данных
  Nil,                  // Пароль
  0,                    // Число дополнительных параметров
  Nil,                  // Перечень полей для доп. параметров
```

```
  Nil,                  // Список доп. параметров
  hDB                   // Дескриптор базы данных
 );
 DbiSetPrivateDir('c:\temp'); // Определение временного каталога
 DbiOpenTable           // Открытие таблицы
 (
  hDB,                  // Дескриптор базы данных
  PChar('COUNTRY'),     // Название таблицы
  PChar(szParadox),     // Тип таблицы (только для локальных БД)
  Nil,                  // Название индекса (необязательный)
  Nil,                  // IndexTagName - только для dBASE
  0,                    // 0 - использовать первичный индекс
  dbiReadWrite,         // Режим редактирования данных
  dbiOpenShared,        // Режим разделения данных
  xltField,             // Режим трансляции данных
  False,                // Признак одностороннего перемещения курсора
  Nil,                  // Дополнительные параметры
  hCur                  // Дескриптор курсора таблицы
 );
 DbiGetCursorProps      // Определение параметров курсора
 (
  hCur,                 // Дескриптор курсора таблицы
  CursProps             // Структура параметров курсора
 );
 GetMem                 // Выделение памяти под буфер записи
(
 RecBuf,
 CursProps.iRecbufSize*SizeOf(Byte)
);
 DbiSetToBegin(hCur);   // Установка курсора в начало набора данных
 DbiGetNextRecord       // Перемещение на первую запись
 (
  hCur,                 // Дескриптор курсора таблицы
  dbiNoLock,            // Режим ограничения доступа
  RecBuf,               // Буфер записи
  Nil                   // Параметры записи
 );
 DbiGetField            // Получение значения поля
```

```
  (
   hCur,                // Дескриптор курсора таблицы
   1,                   // Номер поля в структуре таблицы
   RecBuf,              // Буфер записи
   @FValue,             // Переменная, в которую передается значение
   IsEmpty              // Признак пустой ячейки
  );
  MainForm.CountryEdit.Text := FValue;
  DbiGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
  MainForm.CapitalEdit.Text := FValue;
end;
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  try
  finally
   FreeMem(RecBuf);         // Освобождение памяти буфера записи
   DbiCloseCursor(hCur);    // Закрытие курсора
   DbiCloseDatabase(hDB);   // Закрытие базы данных
   DbiExit;                 // Закрытие сеанса работы с BDE
  end
end;
procedure TMainForm.PriorBtnClick(Sender: TObject);
begin
  try
   if DbiGetPriorRecord(hCur, dbiNoLock, RecBuf, Nil) = DBIERR_BOF
   then PriorBtn.Enabled := False
   else
   begin
    if Not NextBtn.Enabled then NextBtn.Enabled := True;
    DbiGetField(hCur, 1, RecBuf, @FValue, IsEmpty);
    MainForm.CountryEdit.Text := FValue;
    DbiGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
    MainForm.CapitalEdit.Text := FValue;
   end;
  except
   OnBDEError;
  end;
end;
```

```
procedure TMainForm.NextBtnClick(Sender: TObject);
begin
 try
  if DbiGetNextRecord(hCur, dbiNoLock, RecBuf, Nil)=DBIERR_EOF
  then NextBtn.Enabled := False
  else
  begin
   if Not PriorBtn.Enabled then PriorBtn.Enabled := True;
   DbiGetField(hCur, 1, RecBuf, @FValue, IsEmpty);
   MainForm.CountryEdit.Text := FValue;
   DbiGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
   MainForm.CapitalEdit.Text := FValue;
  end;
 except
  OnBDEError;
 end;
end;
end.
```

При показе главной формы приложения в процедуре `FormShow` проводится инициализация BDE, открытие базы данных и таблицы. Пример настроен на работу с демонстрационной базой данных DBDEMOS. При этом создаются дескрипторы базы данных `hDB` и курсора таблицы `hCur`, которые играют в дальнейшей работе приложения важную роль. Если при создании базы данных не указывать псевдоним БД, то затем обязательно нужно определить рабочий каталог базы данных с помощью функции `DbiSetDirectory`.

После этого отводится память под буфер записи, в который будут передаваться значения полей текущей строки таблицы. Размер буфера определяется при помощи структуры `CURProps`.

Затем курсор устанавливается на начало набора данных и на первую запись и осуществляется чтение значений двух полей таблицы.

Навигация по набору данных реализована в методах-обработчиках на нажатие кнопок формы. Их действие аналогично за исключением направления перемещения. При щелчке на кнопке осуществляется переход на следующую или предыдущую запись, данные из новой записи помещаются в буфер записи `RecBuf`. Оттуда при помощи функции `DbiGetField` осуществляется чтение значений полей. При достижении начала или конца набора данных кнопка деактивируется.

При закрытии формы проводятся операции по освобождению памяти буфера записи и закрытию базы данных и BDE.

Соединение с источником данных

Все обращения из приложения к таблицам одной базы данных осуществляются через одно соединение, на которое замыкаются все компоненты доступа к данным, имеющие соответствующие значения свойства `DatabaseName`.

Все управление одиночным соединением с какой-либо базой данных в BDE осуществляется компонентом `TDatabase`. В процессе работы компонент активно использует параметры псевдонимов и драйверов BDE. Свойства и методы компонента `TDatabase` приведены в табл. 20.5.

Таблица 20.5. *Свойства и методы компонента* `TDatabase`

Объявление	Тип	Описание
Свойства		
`property AliasName: string;`	Pb	Задает имя псевдонима BDE используемой базы данных
`property Connected: Boolean;`	Pb	Управляет включением соединения с базой данных
`property DatabaseName: string;`	Pb	Определяет имя базы данных
`property DataSetCount: Integer;`	Ro	Возвращает число открытых наборов данных, работающих через данное соединение
`property DataSets[Index: Integer]: TDBDataSet;`	Ro	Индексированный список всех объектов открытых наборов данных данного соединения
`property Directory: string;`	Pu	Определяет текущий каталог для баз данных Paradox и dBASE
`property DriverName: string;`	Pb	Содержит имя драйвера базы данных
`property Exclusive: Boolean;`	Pb	При значении `True` другие приложения не могут работать с базой данных одновременно с данным компонентом
`type HDBIDB: Longint;` `property Handle: HDBIDB;`	Pu	Дескриптор BDE. Используется для прямых вызовов функций API BDE
`property HandleShared: Boolean;`	Pu	При значении `True` дескриптор BDE компонента доступен в компоненте `TSession`
`property InTransaction: Boolean`	Ro	Показывает состояние транзакции. При значении `True` транзакция выполняется

Таблица 20.5 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`property IsSQLBased: Boolean;`	Ro	При значении `True` соединение работает через драйвер SQL Links
`property KeepConnection: Boolean;`	Pb	При значении `True` соединение продолжает оставаться активным после закрытия всех наборов данных. При значении `False` после закрытия последнего набора данных соединение закрывается
`type TLocale: Pointer;` `property Locale: TLocale;`	Ro	Указывает на языковый драйвер BDE, используемый при работе с базой данных
`property LoginPrompt: Boolean;`	Pb	Управляет отображением стандартного диалога регистрации пользователя при подключении к серверу
`property Params: TStrings;`	Pb	Содержит список значений параметров псевдонима BDE, которые пользователь задает перед подключением к серверу
`property Session: TSession`	Ro	Указывает на компонент `TSession`, который управляет работой данного компонента
`property SessionAlias: Boolean;`	Ro	При значении `True` при подключении к БД используется псевдоним сессии
`property SessionName: string;`	Pb	Содержит имя сеанса, который управляет работой компонента
`property ReadOnly: Boolean;`	Pb	Управляет режимом доступа к данным "только для чтения"
`property Temporary: Boolean;`	Pu	Значение `True` говорит о том, что экземпляр компонента создан во время выполнения

`type`
`  TTraceFlag = (tfQPrepare, tfQExecute, tfError, tfStmt, tfConnect, tfTransact, tfBlob, tfMisc, tfVendor, tfDataIn, tfDataOut);`
`  TTraceFlags = set of TTraceFlag;`
`property TraceFlags: TTraceFlags;`
	Pu	Определяет перечень операций, выполнение которых отображается в утилите SQL Monitor при выполнении приложения

Таблица 20.5 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`type TTransIsolation = (tiDirtyRead, tiReadCommitted, tiRepeatableRead);` `property TransIsolation: TTransIsolation;`	Pb	Определяет уровень изоляции транзакций: `tiDirtyRead` — незавершенное чтение; `tiReadCommitted` — завершенное чтение; `tiRepeatableRead` — повторяемое чтение
Методы		
`procedure ApplyUpdates(const DataSets: array of TDBDataSet);`	Pu	Фиксирует все изменения в наборах данных, работающих через данное соединение, в базе данных
`procedure Close;`	Pu	Закрывает все открытые наборы данных и соединение
`procedure CloseDatasets;`	Pu	Закрывает все открытые наборы данных, работающие через данное соединение
`procedure Commit;`	Pu	Завершает выполнение текущей транзакции и фиксирует все изменения в базе данных
`function Execute(const SQL: string; Params: TParams = nil; Cache: Boolean = False; Cursor: phDBICur = nil): Integer;`	Pu	Выполняет запрос SQL без использования компонента `TQuery`. Текст запроса содержится в параметре SQL. Параметры запроса определяются параметром `Params`. Режим кэширования изменений включается параметром `Cache`. Параметр `Cursor` может применяться при работе с функциями BDE, использующими курсор набора данных
`procedure FlushSchemaCache(const TableName: string);`	Pu	Изменяет представление о структуре таблиц БД, загруженных в память
`procedure Open;`	Pu	Открывает соединение
`procedure Rollback;`	Pu	Отменяет все операции текущей транзакции и завершает ее
`procedure StartTransaction;`	Pu	Начинает выполнение транзакции
`procedure ValidateName(const Name: string);`	Pu	Вызывает исключительную ситуацию, если база данных `Name` уже открыта в текущей сессии

Таблица 20.5 *(окончание)*

Объявление	Тип	Описание
Методы-обработчики событий		
`type TLoginEvent = procedure(Database: TDatabase; LoginParams: TStrings) of object;` `property OnLogin: TLoginEvent;`	Pb	Вызывается при регистрации пользователя на сервере
`property AfterConnect: TNotifyEvent;`	Pb	Вызывается после подключения
`property AfterDisconnect: TNotifyEvent;`	Pb	Вызывается после отключения
`property BeforeConnect: TNotifyEvent;`	Pb	Вызывается перед подключением
`property AfterDisconnect: TNotifyEvent;`	Pb	Вызывается перед отключением

Обычно компонент `TDatabase` размещается в модуле данных приложения.

Для определения базы данных (сервера), с которой приложение устанавливает соединение при помощи компонента `TDatabase`, чаще используется свойство `AliasName`. Свойства `DatabaseName` и `DriverName` предоставляют альтернативный способ создания соединения.

Если соединение задано свойством `AliasName`, то свойство `DatabaseName` пригодно для создания временного псевдонима, который будет доступен только для компонентов доступа к данным внутри приложения. При щелчке на кнопке списка доступных псевдонимов свойства `DatabaseName` в Инспекторе объектов для любого компонента доступа к данным в списке будет доступен и временный псевдоним компонента `TDatabase`.

Например, при переключении приложения на другую базу данных можно изменить только значение псевдонима в компоненте `TDatabase`. Если все компоненты наборов данных подключены к временному псевдониму компонента `TDatabase`, то они автоматически переключатся на новую БД.

Дополнительные возможности управления наборами данных при переключении соединения предоставляют свойства `Connected` и `KeepConnection`. Они позволяют одновременно с соединением закрыть все активные наборы данных.

Если наборы данных приложения подключены к базе данных через компонент TDatabase, то перед их открытием необходимо установить соединение с БД. Соединение с БД устанавливается при помощи метода Open. Если попытаться активизировать набор данных без этого, то соединение будет установлено автоматически.

Аналогичная картина возникает при закрытии наборов данных и отключении от БД. Дополнительное средство управления в этом случае предоставляет свойство KeepConnection. Если оно равно True, то при закрытии последнего открытого набора данных соединение остается открытым. В противном случае соединение автоматически закрывается.

Это позволяет управлять соединением в различных исходных ситуациях. При большой загруженности сервера бывает необходимо прерывать соединение каждый раз. Если требуется разгрузить сетевой трафик, то соединение лучше оставлять включенным.

При подключении к базе данных довольно часто требуется задать значения для параметров драйвера BDE. Для этого есть свойство Params, представляющее собой обычный список. В нем необходимо задавать названия изменяемых параметров и их новые значения:

```
USERNAME=SYSDBA
PASSWORD=masterkey
```

Значения параметров можно задавать как статически, так и динамически во время выполнения.

Компонент TDatabase может облегчить подключение к базам данных с регистрацией пользователей. При регистрации на сервере достаточно задать имя пользователя и пароль в свойстве Params и установить для свойства LoginPrompt значение False. Эта комбинация работает как во время выполнения, так и во время разработки.

Примечание

Для организации доступа к защищенным паролем таблицам Paradox применяется метод AddPassword компонента TSession.

Дополнительные возможности обработки регистрации пользователя дает единственный метод-обработчик OnLogin, программный код которого выполняется вместо появления стандартного диалога ввода имени и пароля. Это позволяет разработчику создавать собственные сценарии регистрации пользователей.

Для обеспечения доступа к функциям API BDE существует свойство Handle (BDE играет важную роль при создании соединения).

Управление выполнением транзакций осуществляется при помощи методов StartTransaction, Commit и RollBack.

Компоненты доступа к данным

Компоненты доступа к данным, необходимые при разработке приложений BDE, располагаются в категории **BDE** Палитры инструментов. Их общими предками являются классы `TBDEDataSet` и `TDBDataSet`. Они обеспечивают работоспособность основных компонентов доступа к данным BDE — `TTable`, `TQuery` и `TStoredProc`.

Класс *TBDEDataSet*

Этот класс является потомком класса `TDataSet`, его значение трудно переоценить: именно `TBDEDataSet` обеспечивает работоспособность важнейших механизмов набора данных за счет обращения к функциям BDE. Например, класс `TBDEDataSet` перекрывает абстрактные методы своего предка `TDataSet`, отвечающие за такие важнейшие операции, как чтение данных и сохранение изменений в базе данных, навигация по записям набора данных, фильтрация.

Напомним, что все эти механизмы не созданы с нуля, а только дополнены обращениями к функциям BDE в необходимых местах методов, изначально описанных в классе `TDataSet`. Например, для обеспечения фильтрации записей набора данных к классу добавлено новое свойство:

```
type
  TFilterOption = (foCaseInsensitive, foNoPartialCompare);
  TFilterOptions = set of TFilterOption;
property FilterOptions: TFilterOptions;
```

Оно определяет дополнительные параметры отбора записей по фильтру (чувствительность к регистру символов и отбор по текстовому шаблону).

Дополнительно к существующим добавлен механизм кэширования изменений. Теперь все вносимые пользователем изменения могут накапливаться в специальном буфере, а их передачей в базу данных можно управлять.

Примечание

Эта возможность очень полезна при создании клиентских приложений в архитектуре "клиент-сервер" и играет ключевую роль при обеспечении возможности редактирования наборов данных сложных запросов SQL.

Дополнительно к методам работы с полями класса `TDataSet` добавлены функции использования полей в формате BLOB.

Для обеспечения функций API BDE на программном уровне добавлено свойство, содержащее дескриптор курсора, соответствующего текущей записи набора данных:

```
type HDBICur: Longint;
property Handle: HDBICur;
```

Также класс обеспечивает возможность программного управления вторичными индексами набора данных в зависимости от типа таблицы базы данных. Полностью свойства и методы класса `TBDEDataSet` приведены в табл. 20.6.

Таблица 20.6. *Свойства и методы класса* `TBDEDataSet`

Объявление	Тип	Описание
Свойства		
`property BlockReadSize: Integer;`	Pu	Определяет размер буфера при блочном чтении данных. Такой режим используется для быстрого перемещения по большим массивам данных. Если значение свойства больше нуля, навигация по набору данных осуществляется без изменения состояния компонентов отображения данных и вызова методов-обработчиков событий
`property CacheBlobs: Boolean;`	Pu	Разрешает использование буфера памяти для данных типа BLOB
`property CachedUpdates: Boolean;`	Pb	Включает или отключает режим кэширования изменений в наборе данных. Используется в клиентских приложениях архитектуры "клиент-сервер"
`property CanModify: Boolean;`	Pu, Ro	Если набор данных позволяет делать изменения, свойство возвращает `True`, иначе возвращается `False`
`property ExpIndex: Boolean;`	Pu, Ro	Показывает, используются ли в наборе данных индексы dBASE
`property Filter: string;`	Pb, Ro	Содержит выражение для фильтра набора данных
`property Filtered: Boolean;`	Pb, Ro	Управляет включением фильтра набора данных
`TFilterOption = (foCaseInsensitive, foNoPartialCompare);` `property FilterOptions: TFilterOptions;`	Pb	Определяет параметры фильтра: `foCaseInsensitive` — строковые значения фильтруются без учета регистра; `FoNoPartialCompare` — при фильтрации символ `*` рассматривается как обычный символ, иначе он означает, что на этом месте может находиться произвольное подмножество любых символов
`type HDBICur: Longint;` `property Handle: HDBICur;`	Pu, Ro	Указатель на курсор BDE, связанный с текущей записью набора данных

Таблица 20.6 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`property KeySize: Word;`	Pu, Ro	Содержит размер ключа для текущего индекса набора данных
`type TLocale: Pointer;` `property Locale: IntPtr;`	Pu, Ro	Указатель на языковый драйвер BDE
`property RecNo: Longint;`	Pu	Номер текущей записи набора данных
`property RecordCount: Longint;`	Ro	Содержит число записей в наборе данных
`property RecordSize: Word;`	Ro	Содержит размер одной записи набора данных
`property UpdateObject: TDataSetUpdateObject;`	Pu	Экземпляр объекта `TUpdateObject`, используемого при кэшировании изменений
`type TUpdateRecordTypes = set of (rtModified, rtInserted, rtDeleted, rtUnmodified);` `property UpdateRecordTypes: TUpdateRecordTypes;`	Pu	Определяет видимость записей в режиме кэширования изменений в зависимости от их состояния: `rtModified` — доступны измененные записи; `rtInserted` — доступны добавленные записи; `rtDeleted` — доступны удаленные записи; `rtUnmodified` — доступны немодифицированные записи
`property UpdatesPending: Boolean;`	Ro	Значение `True` говорит о том, что буфер изменений при кэшировании содержит не сохраненные на сервере изменения
Методы		
`procedure ApplyUpdates;`	Pu	Записывает изменения из буфера в базу данных в режиме кэширования
`function BookmarkValid (Bookmark: TBookmark): Boolean; override;`	Pu	Проверяет существование экземпляра закладки, передаваемого в параметре `Bookmark`

Таблица 20.6 *(продолжение)*

Объявление	Тип	Описание
Методы		
`procedure Cancel;`	Pu	Отменяет все изменения, сделанные в текущей записи с момента последнего сохранения
`procedure CancelUpdates;`	Pu	Отменяет все изменения, сделанные с момента последней записи в базу данных, и очищает буфер в режиме кэширования
`procedure CommitUpdates;`	Pu	Очищает буфер изменений в режиме кэширования
`function CompareBookmarks (Bookmark1, Bookmark2: TBookmark): Integer;`	Pu	Проверяет идентичность закладок, указанных в параметрах `Bookmark1` и `Bookmark2`. При значении 1 сигнализирует о наличии отличий в двух закладках
`function ConstraintCallBack (Req: DsInfoReq; var ADataSources: DataSources): DBIResult; stdcall;`	Pu	Обеспечивает доступ к функциям ограничения данных API BDE
`function ConstraintsDisabled: Boolean;`	Pu	Показывает, включены или отключены ограничения данных
`function CreateBlobStream (Field: TField; Mode: TblobStreamMode): TStream; override;`	Pu	Создает поток для чтения/записи данных типа BLOB
`procedure DisableConstraints;`	Pu	Отключает ограничения данных
`procedure EnableConstraints;`	Pu	Включает ограничения данных
`procedure FetchAll;`	Pu	Переносит все изменения из буфера и восстанавливает все записи от текущей позиции до конца набора данных
`procedure FlushBuffers;`	Pu	Передает в базу данных все изменения из буфера записи
`function GetBlobFieldData (FieldNo: Integer; var Buffer: TblobByteData): Integer; override;`	Pu	Читает все данные BLOB из поля `FieldNo` в буфер `Buffer`

Таблица 20.6 *(продолжение)*

Объявление	Тип	Описание
Методы		
`function GetCurrentRecord(Buffer: PChar): Boolean;`	Pu	Помещает текущую строку в буфер `Buffer`
`procedure GetIndexInfo;`	Pu	Обновляет информацию о текущем индексе набора данных
`function IsSequenced: Boolean; override;`	Pu	Определяет, поддерживает ли таблица БД нумерацию последовательности записей. В классе `TDataSet` всегда возвращает `True`, т. к. абстрактный набор данных свободен от конкретной реализации БД и всегда нумерует записи
`function Locate(const KeyFields: string; const KeyValues: Variant; Options: TlocateOptions): Boolean;`	Pu	Осуществляет поиск в наборе данных. Параметр `KeyFields` содержит список полей, по которым ведется поиск. Параметр `KeyValues` содержит значения полей для поиска. Параметр `Options` определяет условия поиска. Если запись найдена, курсор набора данных устанавливается на эту запись и возвращается `True` *(см. главу 18)*
`function Lookup(const KeyFields: string; const KeyValues: Variant; const ResultFields: string): Variant;`	Pu	Осуществляет поиск в наборе данных. Возвращает массив значений требуемых полей найденной записи. Параметры аналогичны методу `Locate` *(см. главу 18)*
`procedure Post; override;`	Pu	Пересылает сделанные в текущей записи изменения в базу данных
`procedure RevertRecord;`	Pu	Отменяет все изменения в текущей строке при работающем буфере изменений
`procedure Translate(Src, Dest: PChar; ToOem: Boolean); override;`	Pu	Форматирует текст. Если параметр `ToOem == True`, текст `Src` в формате ANSI переводится в текст `Dest` в формате OEM и наоборот
`function UpdateStatus: TUpdateStatus;`		Возвращает тип сохраняемых в буфере изменений данных

Таблица 20.6 *(окончание)*

Объявление	Тип	Описание
Методы-обработчики событий		

`TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);`
 	
`TUpdateErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError; UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object;`
 	
`property OnUpdateError: TUpdateErrorEvent;`
	Ри	Вызывается при возникновении ошибки переноса кэшированных в буфере изменений в таблицу базы данных

`type TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);`
 	
`TUpdateRecordEvent = procedure(DataSet: TDataSet; UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object;`
 	
`property OnUpdateRecord: TUpdateRecordEvent;`
	Ри	Вызывается при сохранении кэшированных в буфере изменений для отдельной записи.
		
		Применяется для организации дополнительного управления этим процессом, например, для контроля какого-либо конкретного значения

Класс *TDBDataSet*

Класс `TDBDataSet` является непосредственным предком основных компонентов доступа к данным `TTable`, `TQuery` и `TStoredProc`. Новые свойства и методы класса обеспечивают соединение набора данных с базой данных и используют функции BDE.

В процессе соединения важнейшую роль играет свойство `DatabaseName`, которое должно содержать псевдоним или полный путь к файлам БД. Для управления отдельным соединением с базой данных можно применять спе-

циальный компонент TDatabase. Указатель на экземпляр такого компонента содержится в свойстве Database.

Многие функции API BDE используют в своей работе дескриптор специальной структуры, описывающей подключенную базу данных. Доступ к этому дескриптору можно получить через свойство DBHandle.

Приложение баз данных одновременно может работать с несколькими наборами данных, каждый из которых подключен к собственной базе данных. Совокупность соединений управляется в рамках сеанса работы, который инкапсулируется компонентом TSession. Указатель на экземпляр такого компонента доступен в наборе данных при помощи свойства DBSession.

Для работы с удаленными серверами в класс введено свойство Provider, обеспечивающее доступ к интерфейсу IProvider. Полностью свойства и методы класса TDBDataSet приведены в табл. 20.7.

Таблица 20.7. *Свойства и методы класса* `TDBDataSet`

Объявление	Тип	Описание
Свойства		
`property AutoRefresh: Boolean;`	Pb	При значении `True` все автоматически создаваемые значения полей (автоинкрементные, значения по умолчанию) обновляются автоматически
`property Database: TDatabase;`	Pu, Ro	Указатель связанного с набором данных компонента `TDatabase`
`property DatabaseName: string;`	Pu, Pb	Псевдоним базы данных
`type HDBISES: Longint;` `property DBHandle: HDBISES;`	Pu, Ro	Дескриптор базы данных. Используется при работе с API BDE
`type TLocale: Pointer;` `property DBLocale: TLocale;`	Pu, Ro	Идентифицирует языковый драйвер API BDE
`property DBSession: TSession`	Pu, Ro	Указатель для компонента `TSession`, с которым работает набор данных
`property Provider: IProvider;`	Pu, Ro	Идентифицирует интерфейс `IProvider`
`property SessionName: string;`	Pu, Ro	Содержит имя компонента сеанса, в котором работает набор данных

Таблица 20.7 *(окончание)*

Объявление	Тип	Описание
Методы		
`function CheckOpen(Status: DBIResult): Boolean;`	Pu	Возвращает результат вызова BDE. Используется для тестирования соединения
`procedure CloseDatabase(Database: TDatabase);`	Pu	Закрывает связь с базой данных, определяемой параметром `Database`
`procedure GetProviderAttributes (List: TList); override;`	Pu	Открывает связь с базой данных, определяемой свойством `DatabaseName`

Компонент *TTable*

Компонент `TTable` инкапсулирует таблицу реляционной базы данных, причем независимо от типа базы данных. Для доступа к данным компонент использует функции BDE.

Необходимая для работы база данных задается свойством `DatabaseName`, в котором можно указать зарегистрированный в BDE псевдоним БД или полный путь к файлам БД.

Таблица БД, на основе которой создается набор данных, определяется свойством `TableName`. При необходимости тип таблицы задается свойством `TableType`, хотя обычно это свойство имеет значение `ttDefault` (см. табл. 20.8), которое включает автоматическое определение типа таблицы по расширению файла.

Примечание

Свойство `TableType` работает только в локальных БД. Обратите внимание, что возможные значения свойства соответствуют основным типам локальных драйверов BDE.

При помощи методов `Open` и `Close` набор данных открывается и закрывается. О его состоянии можно судить по значению свойства `Active`. Более подробно о состоянии набора данных расскажет свойство `State`.

Записи в набор данных можно отбирать при помощи свойств `Filter`, `Filtered` и `FilterOptions`, создающих фильтр, ограничивающий набор данных по значениям данных в одном или нескольких полях.

Методы SetRangeStart, SetRangeEnd, SetRange, ApplyRange, EditRangeStart и EditRangeEnd создают специальный диапазон включаемых в набор данных записей, отбор в диапазон проводится по задаваемым граничным значениям любых полей набора данных.

Поиск нужной записи можно осуществлять методами Lookup или Locate (достаточно просто, но не очень быстро) или через существующие в таблице базы данных индексы — методом FindKey (сложнее, но очень быстро).

От предков компонент унаследовал инструменты для работы с закладками. Это свойство Bookmark и методы GetBookmark, FreeBookmark, GotoBookmark.

Работа с полями осуществляется целой группой свойств и методов, среди которых особое место занимает свойство Fields, представляющее собой индексированный список всех полей набора данных. Это свойство удобно в процессе разработки для организации доступа к полям.

Использование индексов обеспечено свойствами IndexName, IndexFields, IndexFieldNames и IndexFiles.

Свойства MasterSource, MasterField, IndexName дают возможность установить отношение типа "главный—подчиненный" с другой таблицей.

Очень полезны на практике методы и свойства для работы с буфером изменений (свойства CachedUpdates, PendingUpdates, UpdateRecordTypes, методы ApplyUpdates, CancelUpdates, CommitUpdate, RevertRecord). Буфер необходим в клиентских приложениях многоуровневых систем доступа к данным.

От классов TDataSet и TBDEDataSet унаследован обширный набор методов-обработчиков событий, позволяющий решать любые задачи по управлению набором данных.

В табл. 20.8 приведена справочная информация о свойствах и методах компонента TTable.

Таблица 20.8. *Свойства и методы класса* `TTable`

Объявление	Тип	Описание
Свойства		
`property DataSource: TDataSource;`	Pu, Ro	Ссылается на компонент TDataSource главного набора данных в отношении "главный—подчиненный"
`property DefaultIndex: Boolean;`	Pb	Управляет сортировкой данных. При значении True записи упорядочиваются по первичному ключу. При значении False упорядочивание не производится

Таблица 20.8 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`property Exclusive: Boolean;`	Pb	Ограничивает доступ к таблице. При значении `True` с таблицей может работать только одно приложение. Это свойство важно при одновременной работе нескольких приложений с данными в локальной сети
`property Exists: Boolean;`	Pu, Ro	Значение `True` говорит о том, что связанная с компонентом таблица базы данных существует
`property IndexDefs: TindexDefs;`	Pb	Содержит информацию об индексах таблицы
`property IndexFieldCount: Integer;`	Pu, Ro	Возвращает число полей в текущем индексе таблицы
`property IndexFieldNames: string;`	Pb	Разделенный запятыми список названий полей, составляющих текущий индекс. Используется для таблиц серверов SQL
`property IndexFields: [Index: Integer]: TField;`	Pu	Индексированный список полей текущего индекса
`property IndexFiles: TStrings;`	Pb	Список индексных файлов для таблиц dBASE
`property IndexName: string;`	Pb	Определяет вторичный индекс для таблицы. Используется для таблиц локальных СУБД
`property KeyExclusive: Boolean;`	Pu	Управляет границами диапазона, задаваемого методом `SetRange`. При значении `True` крайние записи в диапазон не включаются
`property KeyFieldCount: Integer;`	Pu	Содержит число полей ключа, используемых при поиске. При значении 0 используется только первое поле, при значении 1 используются два первых поля и т. д. По умолчанию устанавливается полное число полей ключа
`property MasterFields: string;`	Pb	Список имен полей главной таблицы, разделенных запятой, используемых при создании отношения "главный—подчиненный"

Таблица 20.8 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`property MasterSource: TDataSource;`	Pb	Содержит имя компонента `TDataSource`, связанного с набором данных, который является главным в отношении "главный—подчиненный"
`property ReadOnly: Boolean;`	Pb	Включает и отключает режим "только для чтения". В некоторых случаях набор данных можно открыть только в этом режиме
`property StoreDefs: Boolean;`	Pb	При значении `True` все сведения об индексах и структуре таблицы хранятся вместе с формой или модулем данных. В этом случае при создании набора данных одновременно создаются поля, индексы, ограничения
`property TableLevel: Integer;`	Pu	Содержит значение уровня таблицы, используемого в драйвере BDE
`property TableName: TFileName;`	Pb	Определяет имя таблицы
`type TTableType = (ttDefault, ttParadox, ttDBase, ttASCII, ttFoxPro);` `property TableType: TTableType;`	Pb	Определяет тип таблицы для стандартного драйвера BDE. Значение `ttDefault` означает, что тип таблицы определяется по расширению файла
Методы		
`procedure AddIndex(const Name, Fields: string; Options: TIndexOptions);`	Pu	Создает новый индекс. Параметр `Name` определяет имя нового индекса, параметр `Fields` — список полей индекса через запятую, параметр `Options` — задает тип индекса
`procedure ApplyRange;`	Pu	Включает в работу границы диапазона, заданные методами `SetRangeStart`, `SetRangeEnd` или `EditRangeStart`, `EditRangeEnd`
`type TBatchMode = (batAppend, batUpdate, batAppendUpdate, batDelete, batCopy);` `function BatchMove (ASource: TBDEDataSet; AMode: TBatchMode): Longint;`	Pu	Переносит записи из таблицы `ASource` в набор данных. Тип операции задается параметром `AMode`. Возвращает число обработанных записей

Таблица 20.8 *(продолжение)*

Объявление	Тип	Описание
Методы		
`procedure CancelRange;`	Pu	Удаляет текущий диапазон
`procedure CloseIndexFile(const IndexFileName: string);`	Pu	Закрывает индексный файл для таблиц dBASE
`procedure CreateTable;`	Pu	Создает новую таблицу, основываясь на данных о структуре таблицы, содержащихся в свойствах `FieldDefs` и `IndexDefs`. Если свойство `FieldDefs` пустое, используется свойство `Fields`. Структура и данные существующей таблицы перезаписываются
`procedure DeleteIndex(const Name: string);`	Pu	Удаляет вторичный индекс
`procedure DeleteTable;`	Pu	Уничтожает таблицу базы данных. Набор данных при этом должен быть закрыт
`procedure EditKey;`	Pu	Переводит набор данных в режим редактирования буфера поиска. После использования этого метода можно изменять значения полей, применяемые для поиска записей
`procedure EditRangeEnd;`	Pu	Разрешает редактирование нижней границы диапазона
`procedure EditRangeStart;`	Pu	Разрешает редактирование верхней границы диапазона
`procedure EmptyTable;`	Pu	Удаляет все записи из набора данных
`function FindKey(const KeyValues: array of const): Boolean;`	Pu	Проводит поиск записи, значения полей которой удовлетворяют условиям, заданным параметром `KeyValues`. Значения разделяются запятыми. Для поиска можно использовать только поля, входящие в текущий индекс. Для локальных стандартных таблиц BDE это поля, определяемые свойством `IndexName`. Для таблиц серверов SQL индекс можно задать свойством `IndexFieldNames`. При успешном поиске функция возвращает `True`

Таблица 20.8 *(продолжение)*

Объявление	Тип	Описание
Методы		
`procedure FindNearest(const KeyValues: array of const);`	Pu	Проводит поиск записи значения полей которой, заданные параметром `KeyValues`, в минимальной степени отличаются от требуемых в большую сторону. Значения для поиска разделяются запятыми. Для поиска можно использовать только поля, входящие в текущий индекс. Для локальных стандартных таблиц BDE это поля, определяемые свойством `IndexName`. Для таблиц серверов SQL индекс можно задать свойством `IndexFieldNames`. При успешном поиске функция возвращает `True`
`procedure GetIndexNames(List: TStrings);`	Pu	Возвращает список индексов таблицы
`procedure GotoCurrent(Table: TTable);`	Pu	Синхронизирует курсор набора данных с курсором таблицы, заданной параметром `Table`
`function GotoKey: Boolean;`	Pu	Устанавливает курсор на запись, соответствующую значениям полей, заданным при последнем применении методов `SetKey` или `EditKey`
`procedure GotoNearest;`	Pu	Устанавливает курсор на запись, точно соответствующую значениям полей, заданным при последнем применении методов `SetKey` или `EditKey`, или ближайшую к ним по значениям в большую сторону
`type TLockType = (ltReadLock, ltWriteLock);` `procedure LockTable (LockType: TLockType);`	Pu	Закрывает доступ к таблице Paradox или dBASE из других приложений
`procedure OpenIndexFile(const IndexFileName: string);`	Pu	Открывает индексный файл таблицы dBASE
`procedure RenameTable(const NewTableName: string);`	Pu	Переименовывает таблицу Paradox или dBASE

Таблица 20.8 *(окончание)*

Объявление	Тип	Описание
Методы		
`procedure SetKey;`	Pu	Очищает буфер поиска. После использования этого метода можно изменять значения полей, используемые для поиска записей
`procedure SetRange (const StartValues, EndValues: array of const);`	Pu	Задает диапазон отбора записей. Параметр `StartValues` определяет значения полей для верхней границы диапазона. Параметр `EndValues` определяет значения полей для нижней границы диапазона. Значения диапазона задаются для полей текущего индекса
`procedure SetRangeEnd;`	Pu	Задает нижнюю границу диапазона. После этого метода необходимо задать значения для полей текущего индекса, которые и будут нижней границей
`procedure SetRangeStart;`	Pu	Задает верхнюю границу диапазона. После этого метода необходимо задать значения для полей текущего индекса, которые и будут верхней границей
`type TLockType = (ltReadLock, ltWriteLock);` `procedure UnlockTable(LockType: TLockType);`	Pu	Разблокирует таблицу Paradox или dBASE для доступа из других приложений

Компонент *TQuery*

Компонент `TQuery` реализует все основные функции стандартного компонента запроса, описанные в *главе 16*. Прямым предком компонента является класс `TDBDataSet`.

Для подключения к базе данных используется свойство `DatabaseName`, в котором задается псевдоним BDE или путь к базе данных.

Текст запроса определяется свойством SQL, для задания которого применяется простой редактор, открывающийся при щелчке на кнопке свойства в Инспекторе объектов (рис. 20.6).

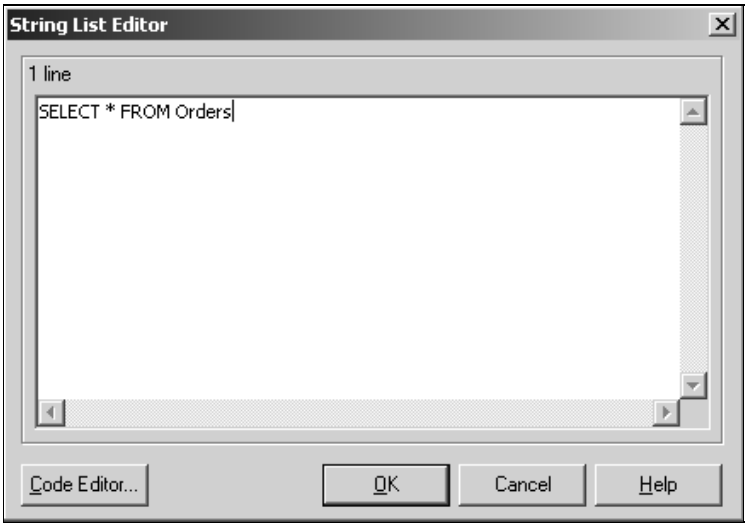


Рис. 20.6. Редактор свойства SQL компонента `TQuery`

Для управления текстом запроса во время выполнения приложения можно использовать возможности класса `TStrings`.

Основные свойства и методы компонента представлены в табл. 20.9.

Таблица 20.9. *Свойства и методы компонента* `TQuery`

Объявление	Тип	Описание
Свойства		
`property Constrained: Boolean;`	Pb	При значении `True` запрещает внесение в набор данных таких значений, которые не соответствуют условиям отбора запроса. Применимо для локальных БД
`property DataSource: TDataSource;`	Pb	Ссылается на компонент `TDataSource`, из набора данных которого задаются значения параметров
`property Local: Boolean;`	Ro	Значение `True` означает, что запрос обращается к локальной таблице
`property ParamCheck: Boolean;`	Pb	При значении `True` параметры запроса обновляются при изменении свойства SQL во время выполнения
`property ParamCount: Word;`	Ro	Возвращает число параметров в запросе

Таблица 20.9 *(окончание)*

Объявление	Тип	Описание
Свойства		
`property Params[Index: Word]TParams;`	Pb	Индексированный список объектов `TParams`, каждый из которых соответствует одному параметру запроса
`property Prepared: Boolean;`	Pu	Возвращает результат выполнения операции подготовки запроса к выполнению
`property RequestLive: Boolean;`	Pb	При значении `False` результат запроса нельзя редактировать, независимо от того, редактируемый результат или нет. При значении `True` результат запроса можно редактировать, но только если он "живой"
`property RowsAffected: Integer;`	Ro	Возвращает число модифицированных записей набора данных с момента последнего выполнения запроса
`property SQL: TStrings;`	Pb	Содержит текст запроса
`property SQLBinary: PChar;`	Pu	Внутреннее свойство для обеспечения работы с BDE
`property StmtHandle: HDBIStmt;`	Ro	Возвращает экземпляр объекта, соответствующего запросу в BDE. Используется при прямом вызове функций BDE
`property Text: PChar;`	Ro	Указатель на символьный массив, содержащий передаваемый в BDE текст запроса
`property UniDirectional: Boolean;`	Pb	Определяет тип используемого курсора данных
Методы		
`procedure ExecSQL;`	Pu	Выполняет запрос без открытия набора данных
`procedure GetDetailLinkFields (MasterFields, DetailFields: TList); override;`	Pu	Заполняет списки параметров метода экземплярами объектов полей двух таблиц запроса, находящихся в отношении "один ко многим"
`function ParamByName(const Value: string): TParam;`	Pu	Возвращает ссылку на экземпляр объекта параметра с именем, переданным в параметре `Value`
`procedure Prepare;`	Pu	Готовит запрос к выполнению
`procedure UnPrepare;`	Pu	Освобождает ресурсы, занятые при подготовке запроса к выполнению

Компонент *TStoredProc*

Компонент `TStoredProc` обеспечивает использование в приложениях BDE хранимых процедур. Прямым предком компонента является класс `TDBDataSet`. Поэтому результатом выполнения хранимой процедуры может быть не только одиночное значение, но и полноценный набор данных.

Основные функции компонента `TStoredProc` соответствуют возможностям стандартного компонента хранимой процедуры, описанного в *главе 16*.

Средствами классов-предков выполняется и подключение компонента к базе данных. Свойство `DatabaseName` определяет базу данных.

Свойство `StoredProcName` задает имя хранимой процедуры.

Перед выполнением хранимую процедуру необходимо подготовить. В частности, на этом этапе осуществляется передача параметров и выделение ресурсов. Эта операция выполняется автоматически методами `ExecProc` и `Open`, или задается явно методом `Prepare`.

Явная подготовка процедуры полезна при неоднократном вызове хранимой процедуры. Если перед первым вызовом процедуры выполнить метод `Prepare`, то все последующие вызовы будут осуществляться без подготовки, которая уже была сделана. В противном случае подготовка будет выполняться автоматически перед каждым вызовом хранимой процедуры. В табл. 20.10 приведены методы и свойства компонента `TStoredProc`.

Таблица 20.10. *Свойства и методы компонента* `TStoredProc`

Объявление	Тип	Описание
Свойства		
`property Overload: Word;`	Pb	Идентификатор процедуры. Используется только для сервера Oracle
`type TParamBindMode = (pbByName, pbByNumber);` `property ParamBindMode: TParamBindMode;`	Pb	Определяет порядок присваивания значений параметров: `pbByName` — по именам параметров; `pbByNumber` — по номерам параметров в списке свойства `Params`
`property ParamCount: Word;`	Ro	Возвращает общее число параметров
`property Params: TParams;`	Pb	Индексированный список параметров
`property Prepared: Boolean;`	Pu	Возвращает `True`, если подготовка процедуры уже проводилась

Таблица 20.10 *(окончание)*

Объявление	Тип	Описание
Свойства		
`property StmtHandle: HDBIStmt;`	Ro	Дескриптор выражения BDE. Используется при прямом вызове функций BDE
`property StoredProcName: string;`	Pb	Содержит имя хранимой процедуры
Методы		
`procedure CopyParams(Value: TParams);`	Pu	Копирует параметры из списка `Value`
`function DescriptionsAvailable: Boolean;`	Pu	При значении `True` параметры хранимой процедуры доступны из приложения
`procedure ExecProc;`	Pu	Передает на сервер сигнал для запуска хранимой процедуры
`procedure GetResults;`	Pu	Возвращает выходные параметры в приложение (используется только для сервера Sybase)
`function ParamByName (const Value: string): TParam;`	Pu	Возвращает параметр с именем `Value`
`procedure Prepare;`	Pu	Готовит процедуру к выполнению
`procedure UnPrepare;`	Pu	Освобождает ресурсы, использованные во время подготовки процедуры

Резюме

Процессор баз данных Borland Database Engine реализует стандартные функции доступа к данным и является ПО промежуточного слоя между приложением и базой данных. При помощи системы драйверов и псевдонимов BDE обеспечивает универсальный способ доступа к данным.

Специальный набор компонентов доступа к данным Delphi использует BDE. Но при необходимости разработчик может обращаться к функциям API BDE напрямую.

Глава 21

Технология dbExpress

Одной из проблем различных технологий доступа к данным, используемым в приложениях Delphi, является трудность распространения готовых приложений. Для BDE требуется отдельная установка, которая занимает порядка 15 Мбайт дискового пространства, а также специальная настройка псевдонимов. ADO предустановлена в операционной системе, но нуждается в настраиваемых провайдерах данных. А при необходимости обновить версию ADO дистрибутив вашего приложения "потяжелеет" более чем на 2 Мбайт.

Технология dbExpress представляет собой совокупность драйверов, компонентов, инкапсулирующих соединения, транзакции, запросы и наборы данных, а также интерфейсов, обеспечивающих универсальный доступ к функциям dbExpress.

Компоненты dbExpress располагаются в Палитре инструментов на одноименной странице.

Технология доступа к данным dbExpress обеспечивает взаимодействие приложения с серверами баз данных на основе специализированных драйверов. Для получения данных драйверы dbExpress применяют только запросы SQL. При этом на клиентской стороне отсутствует кэширование данных, поэтому здесь применяются исключительно однонаправленные курсоры и отсутствует возможность прямого редактирования наборов данных.

Примечание

Проблема редактирования данных в dbExpress может быть решена несколькими путями (см. далее). Однако любые предлагаемые способы повышают затраты на программирование и снижают эффективность полученного кода.

Взамен этих (весьма существенных для построения полноценных приложений) неудобств разработчики получили легкий и быстрый механизм доступа к данным.

Для функционирования компонентов dbExpress необходим только один драйвер, который взаимодействует напрямую с клиентским программным

обеспечением для выбранного сервера БД. В поставку входят драйверы для восьми серверов баз данных:

- DB2
- Informix
- InterBase
- MS SQL Server
- MySQL
- Oracle
- Sybase Adaptive Server Anywhere (ASA)
- Sybase Adaptive Server Enterprise (ASE)

Драйверы реализованы в виде динамических библиотек, а при необходимости могут быть прикомпилированы непосредственно к исполняемому файлу приложения. Поэтому проблема распространения совместно с приложением средств доступа к данным в случае с dbExpress снимается полностью. Естественно, на компьютере должно быть установлено клиентское ПО соответствующего SQL-сервера.

Таким образом, технология dbExpress является наилучшим решением для приложений, в которых необходим быстрый и необременительный просмотр данных серверов SQL, и вряд ли подойдет для сложных клиент-серверных или многозвенных приложений, обеспечивающих серьезную работу с данными.

В этой главе рассматриваются следующие вопросы.

- Настройка соединений с различными серверами баз данных, подключение драйверов и установка их параметров.
- Компоненты dbExpress для просмотра данных и создание пользовательского интерфейса приложений.
- Программная реализация редактирования данных.
- Работа с данными в режиме кэширования изменений и компонент `TSimpleDataSet`.
- Использование интерфейсов.
- Распространение приложений с интегрированной технологией dbExpress.

Доступ к данным dbExpress

Организация доступа к данным в рамках dbExpress обеспечивается тремя составными частями:

- клиентское ПО сервера БД, к которому осуществляется доступ;

- Драйвер dbExpress;
- Компоненты dbExpress в составе приложения, использующего сервер БД.

Схема доступа к данным dbExpress показана на рис. 21.1.

Для организации доступа к данным в первую очередь на компьютере должно быть установлено и настроено клиентское ПО соответствующего сервера БД. Для каждого из четырех поддерживаемых в dbExpress серверов есть особенности установки, обсуждение которых выходит за рамки данной книги.

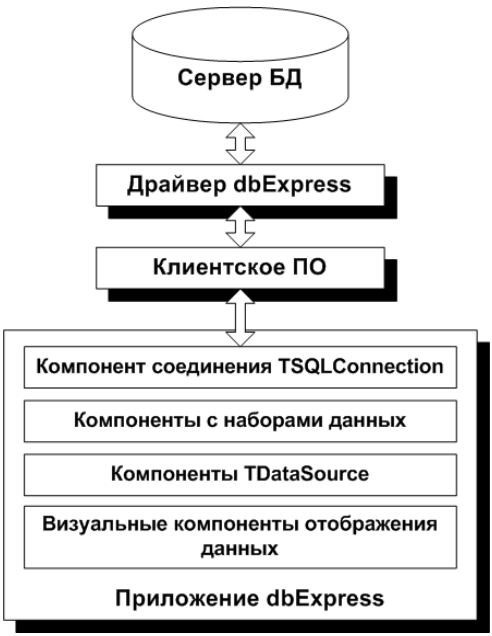


Рис. 21.1. Схема доступа к данным в рамках технологии dbExpress

Для взаимодействия с клиентским ПО сервера технология dbExpress применяет специальные драйверы, которые обеспечивают передачу запросов от приложения.

В остальном приложение не отличается от обычного приложения БД. Приложение должно содержать как минимум один компонент, обеспечивающий соединение с сервером, и необходимое количество компонентов, инкапсулирующих набор данных.

Все необходимые компоненты расположены в категории **dbExpress** Палитры инструментов.

Драйверы доступа к данным

Технология dbExpress обеспечивает доступ к серверу баз данных при помощи драйвера, реализованного в виде динамической библиотеки. Для каждого сервера имеется своя динамическая библиотека (табл. 21.1).

Таблица 21.1. *Драйверы dbExpress*

Сервер БД	Драйвер	Клиентское ПО
DB2	Dbexpdb2.dll	db2cli.dll
Informix	Dbexpinf.dll	isqlb09a.dll
InterBase	Dbexpint.dll	gds32.dll
MS SQL Server	Dbexpmss.dll	OLE DB
MySQL	Dbexpmy.dll	libmysql.dll
Oracle	Dbexpora.dll	oci.dll
Sybase ASA	Dbexpasa.dll	Dbodbc9.dll
Sybase ASE	Dbexpase.dll	libct.dll; libcs.dll

Перечисленные в табл. 28.1 файлы драйверов находятся в папке ..\Bin.

Для доступа к данным сервера драйвер должен быть установлен на компьютере клиента. Драйвер взаимодействует с клиентским ПО сервера, которое также должно быть инсталлировано на клиентской стороне.

Стандартные настройки для каждого драйвера хранятся в файле ..\Program Files\Common Files\Borland Shared\BDS\DBExpress\dbxdrivers.ini.

Соединение с сервером баз данных

Создание соединения приложения с сервером в рамках технологии dbExpress осуществляется через компонент `TSQLConnection`. Это обязательный компонент, все остальные компоненты связаны с ним и используют его для получения данных.

После переноса этого компонента в модуль данных или на форму необходимо выбрать тип сервера и настроить параметры соединения.

Свойство

```
property ConnectionName: string;
```

позволяет выбрать из выпадающего списка конкретное настроенное соединение. По умолчанию разработчику доступно по одному настроенному со-

единению для каждого сервера БД. После выбора соединения автоматически устанавливаются значения следующих свойств (см. табл. 21.1).

Свойство

```
property DriverName: string;
```

определяет используемый драйвер

Свойство

```
property LibraryName: string;
```

задает динамическую библиотеку драйвера dbExpress

Свойство

```
property VendorLib: string;
```

определяет динамическую библиотеку клиентского ПО сервера (см. табл. 21.1).

Свойство

```
property Params: TStrings;
```

инкапсулирует параметры соединения, в списке которого содержатся настройки для выбранного соединения.

При необходимости все перечисленные свойства можно установить дополнительно.

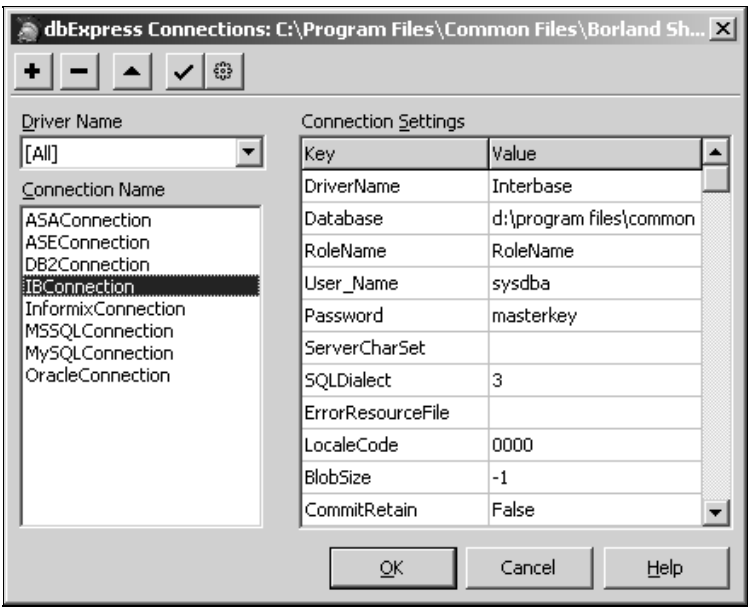


Рис. 21.2. Окно редактора настроенных соединений компонента `TSQLConnection`

Разработчик может дополнять и изменять список настроенных соединений. Для этого есть специализированный Редактор соединений **dbExpress Coonections** (рис. 21.2). Он открывается после двойного щелчка на компоненте `TSQLConnection` или выбора команды **Edit Connection Properties** из всплывающего меню компонента.

В списке слева располагаются существующие соединения. В правой части для выбранного соединения отображаются текущие настройки. При помощи кнопок в панели инструментов можно создавать, переименовывать и удалять соединения. Настройки также можно редактировать.

Список соединений в левой части соответствует списку выбора свойства `ConnectionName` в Инспекторе объектов. Настройки соединения из правой части отображаются в свойстве `Params`.

Настроенные соединения и их параметры сохраняются в файле ..\Program Files\Common Files \Borland Shared\BDS\DBExpress\dbxconnections.ini. Конкретные значения настроек соединений зависят от сервера БД и требований приложения (табл. 21.2).

Таблица 21.2. *Настройка соединения dbExpress*

Параметр	Значение
Общие настройки	
`BlobSize`	Задает ограничение на объем пакета данных для данных BLOB
`DriverName`	Имя драйвера
`ErrorResourceFile`	Файл сообщений об ошибках
`HostName`	Имя компьютера, на котором работает сервер
`LocaleCode`	Код локализации, определяющий влияние национальных символов на сортировку данных
`User_Name`	Имя пользователя
`Password`	Пароль
DB2	
`Database`	Имя клиентского ядра
`DB2 TransIsolation`	Уровень изоляции транзакций
Informix	
`Database`	Имя базы данных
`Informix TransIsolation`	Уровень изоляции транзакций
`Trim Char`	Признак разделителя символов

Таблица 21.2 *(продолжение)*

Параметр	Значение
InterBase	
`CommitRetain`	Задает поведение курсора после завершения транзакции. При значении `True` курсор обновляется, иначе — удаляется
`Database`	Имя файла базы данных (файл с расширением GDB)
`InterBase TransIsolation`	Уровень изоляции транзакции
`RoleName`	Роль пользователя
`ServerCharSet`	Кодировка символов на сервере
`SQLDialect`	Используемый диалект SQL. Для InterBase 5 возможно только одно значение — 1
`WaitOnLocks`	Разрешение на ожидание занятых ресурсов
MS SQL	
`Database`	Имя базы данных
`MS SQL TransIsolation`	Уровень изоляции транзакций
`OS Autentification`	Признак регистрации пользователя на сервере на основе учетной записи в операционной системе
`Prepare SQL`	Признак предварительной подготовки запросов SQL
MySQL	
`Database`	Имя базы данных
Oracle	
`AutoCommit`	Флаг завершения транзакции. Устанавливается только сервером
`BlockingMode`	Задает режим завершения запроса. При значении `True` соединение дожидается окончания запроса (синхронный режим), иначе — начинает выполнение следующего (асинхронный режим)
`Database`	Запись базы данных в файле TNSNames.ora
`Oracle TransIsolation`	Уровень изоляции транзакций
Sybase ASA	
`ConnectionString`	Флаг завершения транзакции. Устанавливается только сервером

Таблица 21.2 (окончание)

Параметр	Значение
Sybase ASA	
`ASA TransIsolation`	Уровень изоляции транзакций
Sybase ASE	
`ASE TransIsolation`	Уровень изоляции транзакций

После выбора настроенного соединения или выбора типа сервера и самостоятельной настройки параметров соединения компонент `TSQLConnection` готов к работе.

Свойство

```
property Connected: Boolean;
```

открывает соединение с сервером при значении `True`. Аналогичную операцию выполняет метод

```
procedure Open;
```

После открытия соединения все компоненты dbExpress, инкапсулирующие наборы данных и связанные с открытым компонентом `TSQLConnection`, получают доступ к базе данных.

Соединение закрывается тем же свойством `Connected = False` или методом

```
procedure Close;
```

При открытии и закрытии соединения разработчик может использовать обработчики событий, вызывающиеся до и после открытия и закрытия соединения соответственно

```
property BeforeConnect: TNotifyEvent;
property BeforeDisconnect: TNotifyEvent;
property AfterConnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
```

Например, на стороне клиента можно организовать проверку пользователя приложения:

```
procedure TForm1.MyConnectionBeforeConnect(Sender: TObject);
begin
 if MyConnection.Params.Values['User_Name']) <> DefaultUser then
 begin
  MessageDlg('Wrong user name', mtError, [mbOK], 0);
```

```
  Abort;
 end;
end;
```

Свойство

```
property LoginPrompt: Boolean;
```

определяет, нужно ли отображать диалог авторизации пользователя перед открытием соединения. При значении `True` диалог отображается.

О текущем состоянии соединения можно судить по значению свойства

```
TConnectionState = (csStateClosed, csStateOpen, csStateConnecting,
csStateExecuting, csStateFetching, csStateDisconnecting);
property ConnectionState: TConnectionState;
```

Параметры соединения можно настраивать на этапе разработки в Инспекторе объектов или Редакторе соединений (см. рис. 21.2). Также это можно сделать и непосредственно перед открытием соединения, используя свойство `Params` или метод

```
procedure LoadParamsFromIniFile( AFileName : String = '' );
```

который загружает заранее подготовленные параметры из INI-файла.

Проверить успешность этой операции можно при помощи свойства

```
property ParamsLoaded: Boolean;
```

значение `True` которого сигнализирует об успехе загрузки (листинг 21.1).

Листинг 21.1. Пример загрузки параметров соединения

```
procedure TForm1.StartBtnClick(Sender: TObject);
begin
 if MyConnection.Params.Values['DriverName'] = '' then
 MyConnection.LoadParamsFromIniFile('c:\Temp\dbxalarmconnections.ini');
 if MyConnection.ParamsLoaded then
 try
  MyConnection.Open;
 except
  MessageDlg('Database connection error', mtError, [mbOK], 0);
 end;
end;
```

Управление наборами данных

Дополнительно компонент TSQLConnection позволяет выполнять некоторые операции с подключенными наборами данных и следить за их состоянием. Рассмотрим их.

Свойство

```
property DataSetCount: Integer;
```

возвращает число подключенных через данное соединение наборов данных.

Но это только активные наборы данных, переданные в связанные компоненты. Общее число выполняющихся в данный момент запросов возвращает свойство

```
property ActiveStatements: LongWord;
```

Если сервер БД установил для данного соединения максимальное число одновременно выполняющихся запросов, то оно доступно в свойстве

```
property MaxStmtsPerConn: LongWord;
```

Поэтому перед открытием набора данных можно выполнять следующий код, который повысит надежность приложения:

```
if MyQuery.SQLConnection.ActiveStatements <=
   MyQuery.SQLConnection.MaxStmtsPerConn
    then MyQuery.Open
    else MessageDlg('Database connection is busy', mtWarning, [mbOK], 0);
```

В случае возникновения непредвиденной ситуации все открытые через данное соединение наборы данных можно быстро закрыть методом

```
procedure CloseDataSets;
```

без разрыва соединения.

При необходимости компонент TSQLConnection может самостоятельно выполнять запросы SQL, не прибегая к помощи компонентов TSQLQuery или TSQLDataSet. Для этого предназначена функция

```
function Execute(const SQL: string; Params: TParams);
ResultSet:Pointer=nil): Integer;
```

Если запрос должен содержать параметры, то необходимо сначала создать объект списка параметров TParams и заполнить его. При этом, т. к. объект TParams еще не связан с конкретным запросом, важен порядок следования параметров, который должен совпадать в списке TParams и в тексте SQL.

Если запрос возвращает результат, метод автоматически создает объект типа TCustomSQLDataSet и возвращает указатель на него в параметр ResultSet.

Функция возвращает число обработанных запросом записей. Следующий фрагмент кода (листинг 21.2) иллюстрирует применение функции `Execute`.

Листинг 21.2. Применение функции `Execute`

```
procedure TForm1.SendBtnClick(Sender: TObject);
var FParams: TParams;
    FDataSet: TSQLDataSet;
begin
 FParams := TParams.Create;
 try
  FParams.Items[0].AsInteger := 1234;
  FParams.Items[1].AsInteger := 6751;
  MyConnection.Execute('SELECT * FROM Orders WHERE OrderNo >= :Ord AND
EmpNo = :Emp', FParams, FDataSet);
  if Assigned(FDataSet) then
  with FDataSet do
  begin
   Open;
   while Not EOF do
   begin
    {...}
    Next;
   end;
   Close;
  end;
 finally
  FParams.Free;
 end;
end;
```

Если запрос не имеет настраиваемых параметров и не возвращает набор данных, можно использовать функцию

```
function ExecuteDirect(const SQL: string ): LongWord;
```

которая возвращает 0 в случае успешного выполнения запроса или код ошибки.

Перегружаемый метод

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
procedure GetTableNames(List: TStrings; SchemaName: string; SystemTables:
Boolean = False);
```

возвращает список таблиц базы данных. Параметр `SystemTables` позволяет включать в формируемый список `List` системные таблицы.

Метод `GetTableNames` дополнительно управляется свойством

```
TTableScope = (tsSynonym, tsSysTable, tsTable, tsView);
TTableScopes = set of TTableScope;
property TableScope: TTableScopes;
```

которое позволяет задать тип таблиц, имена которых попадают в список.

Для каждой таблицы можно получить список полей, применив перегружаемый метод

```
procedure GetFieldNames(const TableName: String; List: TStrings);
procedure GetFieldNames(const TableName: String; SchemaName: string;
List: TStrings);
```

и список индексов при помощи перегружаемого метода

```
procedure GetIndexNames(const TableName: string; List: TStrings);
procedure GetIndexNames(const TableName: string; SchemaName: string;
List: TStrings);
```

В обоих методах список возвращаемых значений содержится в параметре `List`.

Аналогичным образом перегружаемый метод

```
procedure GetProcedureNames(List: TStrings);
procedure GetProcedureNames(const PackageName: string; List: TStrings);
procedure GetProcedureNames(const PackageName: string; const SchemaName:
string; List: TStrings);
```

возвращает список доступных хранимых процедур, а перегружаемый метод

```
procedure GetProcedureParams(ProcedureName: String; List: TList);
procedure GetProcedureParams(ProcedureName: String; PackageName: string;
List: TList);
procedure GetProcedureParams(ProcedureName: String; PackageName: string;
SchemaName: string; List: TList);
```

определяет параметры отдельной процедуры.

Транзакции

Подобно своим аналогам в BDE и ADO компонент `TSQLConnection` поддерживает механизм транзакций и делает это сходным образом.

Начало, фиксацию и откат транзакции выполняют методы

```
procedure StartTransaction(TransDesc: TTransactionDesc);
procedure Commit(TransDesc: TTransactionDesc);
procedure Rollback(TransDesc: TTransactionDesc);
```

При этом запись `TtransactionDesc` возвращает параметры транзакции:

```
TTransIsolationLevel = (xilDIRTYREAD, xilREADCOMMITTED,
xilREPEATABLEREAD, xilCUSTOM);
  TTransactionDesc = packed record
    TransactionID    : LongWord;
    GlobalID         : LongWord;
    IsolationLevel   : TTransIsolationLevel;
    CustomIsolation  : LongWord;
  end;
```

Запись содержит уникальный в рамках соединения идентификатор транзакции `TransactionID` и уровень изоляции транзакции `IsolationLevel`. При уровне изоляции `xilCUSTOM` определяется параметр `CustomIsolation`. Идентификатор `GlobalID` служит для работы с сервером Oracle.

Некоторые серверы БД не поддерживают транзакции, и для определения этого факта имеется свойство

```
property TransactionsSupported: LongBool;
```

Если соединение уже находится в транзакции, свойству

```
property InTransaction: Boolean;
```

присваивается значение `True`. Поэтому если сервер не поддерживает множественные транзакции, всегда полезно убедиться, что соединение не обслуживает начатую транзакцию:

```
var TransInfo: TTransactionDesc;
{...}
if Not MyConnection.InTransaction then
 try
  MyConnection.StartTransaction(TransInfo);
  {...}
  MyConnection.Commit(TransInfo);
 except
  MyConnection.Rollback(TransInfo);
 end;
```

Использование компонентов наборов данных

Набор компонентов dbExpress, инкапсулирующих набор данных, вполне обычен и сравним с аналогичными компонентами BDE, dbGo и InterBase Express. Это компоненты `TSQLDataSet`, `TSQLTable`, `TSQLQuery` и `TSQLStoredProc`.

Примечание

Компонент `TSimpleDataSet` также относится к рассматриваемой группе, но поскольку он обладает рядом специфических возможностей, его описание вынесено в отдельный пункт.

Однако необходимость создания простой технологии доступа к данным, какой является dbExpress, наложила на эти компоненты ряд ограничений.

Хотя общим предком всех рассматриваемых здесь компонентов является класс `TDataSet`, который обладает полным инструментарием для работы с набором данных, компоненты dbExpress работают только с однонаправленными курсорами и не позволяют редактировать данные. Однонаправленные курсоры ограничивают навигацию по набору данных и обеспечивают перемещение только на следующую запись и возврат на первую. Также здесь недоступны любые операции, требующие буферизации данных, поиск, фильтрация и синхронный просмотр.

Таким образом, для выключения ряда механизмов класса `TDataSet` понадобился еще один промежуточный класс `TCustomSQLDataSet`.

Отображение данных при помощи компонентов из категории **Data Controls** Палитры инструментов также ограничено. Недопустимы компоненты `TDBGrid` и `TDBCtrlGrid`, а при использовании `TDBNavigator` следует отключить кнопки возврата на одну позицию назад и перехода на последнюю запись. Также ничего хорошего не получится из попытки применить компоненты синхронного просмотра. Остальные компоненты можно использовать обычным способом.

Доступ к данным во всех рассматриваемых компонентах осуществляется одинаково: через соединение, инкапсулированное компонентом `TSQLConnection`. Привязку к соединению выполняет свойство

```
property SQLConnection: TSQLConnection;
```

Рассмотрим теперь компоненты dbExpress подробнее.

Класс *TCustomSQLDataSet*

Так как общим предком компонентов dbExpress является класс `TDataSet`, то задача класса `TCustomSQLDataSet` — не столько внесение новой функцио-

нальности, сколько корректное ограничение возможностей, заложенных в `TDataSet`. Непосредственно в приложениях этот класс не используется, но информация о нем полезна для понимания других компонентов dbExpress и для создания собственных компонентов на его основе.

Класс `TCustomSQLDataSet` — общий предок для компонентов, инкапсулирующих запросы, таблицы и хранимые процедуры. Для их поддержки имеются следующие два свойства:

- `TSQLCommandType = (ctQuery, ctTable, ctStoredProc);`

  `property CommandType: TSQLCommandType;`

  определяющие тип команды, направляемой серверу;
- `property CommandText: string;`

содержащее текст команды.

Если серверу передается запрос SQL (`CommandType = ctQuery`), свойство `CommandText` содержит текст запроса. Если это команда на получение таблицы, свойство `CommandText` содержит имя таблицы, а далее с использованием имени таблицы автоматически создается запрос SQL на получение всех полей этой таблицы. Если необходимо выполнить процедуру, свойство `CommandText` содержит имя этой процедуры.

Текст команды, которая реально передается на сервер для выполнения, содержится в защищенном свойстве

```
property NativeCommand: string;
```

Для табличного представления существует свойство

```
property SortFieldNames: string;
```

определяющее порядок сортировки записей табличного набора данных. Свойство должно содержать список полей, разделенных точкой с запятой. Это свойство необходимо для создания выражения ORDER BY для генерируемой команды.

Для обработки исключительных ситуаций в классах-потомках может быть использовано защищенное свойство

```
property LastError: string;
```

которое возвращает текст последней ошибки dbExpress.

Для ускорения работы набора данных можно отключить получение от сервера метаданных об объекте запроса, которые обычно направляются клиенту вместе с результатом запроса. Для этого свойству

```
property NoMetadata: Boolean;
```

присваивается значение `True`. Однако пользоваться им нужно осторожно, т. к. для некоторых видов команд метаданные необходимы (операции с использованием индексов).

Разработчик может управлять процессом получения метаданных. Для этого необходимо заполнить структуру

```
TSchemaType = (stNoSchema, stTables, stSysTables, stProcedures,
stColumns, stProcedureParams, stIndexes );

  TSQLSchemaInfo = record
    FType      : TSchemaType;
    ObjectName : String;
    Pattern    : String;
    PackageName : string;
  end;
```

которая доступна через защищенное свойство

```
property SchemaInfo: TSQLSchemaInfo;
```

а значит, пригодна только при создании новых компонентов на основе `TCustomSQLDataSet`.

Параметр `FType` определяет тип требуемой информации. Параметр `ObjectName` определяет имя таблицы или хранимой процедуры, если в параметре `FType` указаны поля, индексы или параметры процедур.

> **Внимание**
>
> Если компонент должен получать результирующий набор данных, параметр `FType` должен обязательно иметь значение `stNoSchema`. При изменении значения свойства `CommandText` это условие выполняется автоматически.

Параметр `Pattern` определяет, какие ограничения накладываются на метаданные. Он содержит символьную маску, подобную свойству `Mask` многих визуальных компонентов. Последовательность символов маски обозначается символом `%`, единичный символ определяется символом `_`.

При необходимости использовать управляющие символы в качестве маскирующих применяются двойные символы `%%` и `__`.

Подобно свойству `Tag` класса `TComponent` класс `TCustomSQLDataSet` имеет строковое свойство

```
property DesignerData: string
```

в котором разработчик может хранить любую служебную информацию. По существу, это просто лишняя строковая переменная, которую нет необходимости объявлять.

Компонент *TSQLDataSet*

Компонент `TSQLDataSet` является универсальным и позволяет выполнять запросы SQL (подобно `TSQLQuery`), просматривать таблицы целиком (подобно `TSQLTable`) или выполнять хранимые процедуры (подобно `TSQLStoredProc`).

Режим работы компонента определяется свойством `CommandType` (см. ранее).

Если ему присвоить значение `ctTable`, в списке свойства `CommandText` можно выбрать имя таблицы, если, конечно, компонент подключен к соединению. При выборе значения `ctQuery` в свойстве `CommandText` необходимо определить текст запроса SQL. Для работы в режиме хранимой процедуры для свойства `CommandType` устанавливается значение `ctStoredProc`, а в списке свойства `CommandText` можно выбрать нужную процедуру.

Для открытия набора данных существуют традиционные способы: свойство `Active` или метод `Open`. Если же запрос SQL или хранимая процедура не возвращают набор данных, для их выполнения применяется метод

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Параметр `ExecDirect` определяет, необходимо ли выполнять подготовку параметров перед выполнением команды. Если параметры запроса или процедуры существуют, параметр `ExecDirect` должен иметь значение `False`.

Дополнительно для табличного режима можно использовать свойство `SortFieldNames`, определяющее порядок сортировки записей таблицы.

В режиме запросов и хранимых процедур параметры задаются свойствами `Params` и `ParamCheck`.

Информация об используемых в результирующем наборе данных индексах сохраняется в свойстве

```
property IndexDefs: TIndexDefs;
```

Подробнее о типе `TIndexDefs` *см. главу 16.*

Компонент *TSQLTable*

Компонент `TSQLTable` предназначен для просмотра таблиц целиком и по основным функциям подобен своим аналогам `TTable`, `TADOTable` и `TIBTable`. Подробно о работе с табличными компонентами рассказано в *главе 20.*

Для определения имени таблицы имеется свойство

```
property TableName: string;
```

и если компонент подключен к соединению, имя таблицы можно выбрать из списка. После определения имени таблицы и настройки соединения

компонент уже готов к работе. После его открытия стандартными средствами (свойством `Active` или методом `Open`) в компонент передается набор данных из выбранной таблицы.

Для получения табличного набора данных компонент `TSQLTable` самостоятельно формирует запрос на сервер, используя для этого возможности, унаследованные от предка `TCustomSQLDataSet`.

Метод

```
procedure PrepareStatement; override;
```

генерирует для выбранной таблицы текст запроса, который формируется компонентом для передачи на сервер.

Преимуществом всех табличных компонентов является возможность работать с индексами. Для подключения простых или составных индексов служат свойства `IndexFieldNames`, `IndexFields`, `IndexName`. А метод

```
procedure GetIndexNames(List: TStrings);
```

возвращает в параметр `List` список используемых индексов.

Связь между двумя наборами данных "главный—подчиненный" организуется свойствами `MasterFields`, `MasterSource`.

Компонент `TSQLTable` предоставляет разработчику некоторое подобие функций редактирования. Для удаления всех записей из связанной с компонентом таблицы на сервере существует метод

```
procedure DeleteRecords;
```

Компонент *TSQLQuery*

Компонент `TSQLQuery` позволяет выполнять на сервере запросы SQL клиента. Все его возможности совпадают со стандартными для компонентов данного типа *(см. главу 16)*.

Настройка на соединение с сервером осуществляется свойством

```
property SQLConnection: TSQLConnection;
```

в котором выбирается компонент соединения `TSQLConnection`.

Текст запроса содержится в свойстве

```
property SQL: TStrings;
```

а его простое строковое представление в свойстве

```
property Text: string;
```

Редактирование текста запроса выполняется в стандартном редакторе, открывающемся при щелчке на кнопке свойства в Инспекторе объектов.

Параметры запроса должны содержаться в индексированном свойстве

```
property Params: TParams;
```

Подготовка параметров и запроса в целом на сервере осуществляется свойством

```
property Prepared: Boolean;
```

Если оно имеет значение `True`, для запроса на сервере выделяются ресурсы, что ускоряет его выполнение:

```
...
SQLQuery.Prepared := False;
SQLQuery.ParamByName('SomeParam1').AsString := 'SomeValue';
SQLQuery.ParamByName('SomeParam2').Clear;
SQLQuery.Prepared := True;
...
```

Свойство

```
property ParamCheck: Boolean;
```

определяет, будет ли пересоздаваться список параметров запроса, если его текст был изменен. Например, запрос применяется для создания хранимых процедур, исходный текст которых передается на сервер в запросе. В теле хранимой процедуры могут содержаться собственные параметры, которые при анализе запроса могут быть интерпретированы как параметры самого запроса. В этом случае, конечно, возникнет ошибка создания параметров. Здесь как раз очень уместно свойство `ParamCheck`:

```
...
SQLQuery.ParamCheck := False;
 SQLQuery.SQL.Clear;
 SQLQuery.SQL.Add('SomeSQLTextForNewStoredProc');
 SQLQuery.ExecSQL();
...
```

Если запрос возвращает набор данных, его выполнение осуществляется свойством `Active` или методом `Open`. В противном случае — методом

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Параметр `ExecDirect = False` означает, что запрос не имеет настраиваемых параметров.

Компонент *TSQLStoredProc*

Компонент `TSQLStoredProc` инкапсулирует функциональность хранимых процедур для их выполнения в рамках технологии dbExpress. Все его функ-

ции стандартны. Подробнее о функциях компонентов хранимых процедур *см. главу 16.*

Настройка на соединение с сервером осуществляется свойством

```
property SQLConnection: TSQLConnection;
```

в котором выбирается компонент соединения `TSQLConnection`.

Имя хранимой процедуры определяется свойством

```
property StoredProcName: string;
```

Если соединение с сервером уже настроено, имя хранимой процедуры может быть выбрано из выпадающего списка свойства в Инспекторе объектов.

Для работы с входными и выходными параметрами предназначено свойство

```
property Params: TParams;
```

Внимание

При работе с параметрами желательно обращаться к конкретному параметру по имени при помощи метода `ParamByName`. При работе с некоторыми серверами порядок следования параметров до выполнения процедуры и после может изменяться.

Свойство

```
property ParamCheck: Boolean;
```

определяет, будет ли пересоздаваться список параметров, если хранимая процедура была изменена. Для этого свойство должно иметь значение `True`.

Процедура выполняется методом

```
function ExecProc: Integer; virtual;
```

если она не возвращает набор данных. Иначе используются свойство `Active` или метод `Open`.

Если хранимая процедура возвращает несколько связанных наборов данных (подобно иерархическим запросам ADO), доступ к следующему набору данных осуществляет метод

```
function NextRecordSet: TCustomSQLDataSet;
```

автоматически создавая объект типа `TCustomSQLDataSet` для инкапсуляции новых данных. Возврат к предыдущему набору данных возможен, если вы определили объектные переменные для каждого набора данных (листинг 21.3).

Листинг 21.3. Применение `TCustomSQLDataSet`

```
var SecondSet: TCustomSQLDataSet;
...
```

```
MyProc.Open;
while Not MyProc.Eof do
begin
 {...}
 Next;
end;
SecondSet := MyProc.NextRecordSet;
SecondSet.Open;
{...}
SecondSet.Close;
MyProc.Close;
```

Компонент *TSimpleDataSet*

Компонент `TSimpleDataSet` обеспечивает кэширование полученных данных и сделанных изменений на стороне клиента и последующую передачу их на сервер для фиксации. В отличие от компонента `TClientDataSet`, основным назначением которого является обслуживание набора данных, полученного от удаленного сервера при помощи серверных компонентов `DataSnap`, компонент `TSimpleDataSet` призван быть лишь средством редактирования набора данных в технологии dbExpress.

Компонент работает с двунаправленным курсором и позволяет редактировать данные, правда только в режиме кэширования.

Таким образом, компонент `TSimpleDataSet` позволяет исправить основные недостатки технологии dbExpress.

Для подключения к источнику данных компонент использует свойство

```
property DBConnection: TSQLConnection;
```

которое позволяет связать его с соединением `TSQLConnection` (см. ранее в этой главе) или свойство

```
property ConnectionName: string;
```

которое позволяет выбрать тип соединения dbExpress напрямую.

При этом у компонента отсутствует механизм создания удаленного доступа к данным, представленный у компонента `TClientDataSet` свойствами `RemoteServer` и `ProviderName`.

После создания соединения с сервером БД можно определить тип команды, подобно компоненту `TSQLDataSet`.

Тип команды определяется свойством

```
TSQLCommandType = (ctQuery, ctTable, ctStoredProc);
property CommandType: TSQLCommandType;
```

А содержание команды задает свойство

```
property CommandText: string;
```

После этого компонент можно связывать с компонентами отображения данных, просматривать и редактировать данные.

Для передачи на сервер сделанных и сохраненных в локальном кэше изменений есть метод

```
function ApplyUpdates(MaxErrors: Integer); Integer; virtual;
```

где параметр `MaxErrors` определяет максимально возможное число ошибок при сохранении. Обычно этому параметру присваивается −1, что снимает ограничение на число ошибок.

Метод

```
function Reconcile(const Results: OleVariant): Boolean;
```

очищает локальный кэш компонента от записей, которые успешно сохранены на сервере.

Отменить локальные изменения можно методом

```
procedure CancelUpdates;
```

Обратите внимание, что в компоненте действуют традиционные методы набора данных `Edit`, `Post`, `Cancel`, `Apply`, `Insert` и `Delete`. Но они оказывают влияние только на записи, кэшированные локально. Вы можете сколько угодно редактировать набор данных при помощи перечисленных методов, но они будут изменять только содержимое кэша. Настоящее сохранение на сервере осуществляется методом `ApplyUpdates`.

Данные между сервером и компонентом пересылаются пакетами.

Доступ к текущему пакету возможен при помощи свойства

```
property Data: OleVariant;
```

А сделанные изменения содержатся в свойстве

```
property Delta: OleVariant;
```

При этом разработчик может регулировать размер пакетов. Например, при ухудшении соединения можно уменьшить размер пакетов. Размер пакета определяется свойством

```
property PacketRecords: Integer;
```

которое задает число записей в пакете. Автоматическое назначение пакетов включается при

```
PacketRecords := -1
```

Если значение `PacketRecords` равно 0, между клиентом и сервером пересылаются только метаданные.

Если свойство `PacketRecords` больше нуля, то необходимо вручную организовывать подкачку данных с сервера. Для этого применяется метод

```
function GetNextPacket: Integer;
```

А для организации такой подкачки вполне подойдут методы-обработчики событий

```
property BeforeGetRecords: TRemoteEvent;
property AfterGetRecords: TRemoteEvent;
```

В компоненте `TSimpleDataSet` развиты средства работы с одиночными записями. Можно просмотреть общее число записей

```
property RecordCount: Integer;
```

и номер текущей записи

```
property RecNo: Integer;
```

Размер одной записи сохраняется в свойстве

```
property RecordSize: Word;
```

Все изменения, сделанные в текущей записи, отменяются методом

```
procedure RevertRecord;
```

а обновить значение полей для текущей записи с сервера можно методом

```
procedure RefreshRecord;
```

Обработка исключительных ситуаций для компонента `TSimpleDataSet` состоит из двух этапов.

Во-первых, необходимо отслеживать ошибки на стороне клиента (некорректный ввод данных, ошибки кэширования и т. д.). В этом случае подходят все стандартные способы, применяемые для наборов данных.

Во-вторых, ошибки могут возникнуть при сохранении изменений на сервере. И поскольку само событие, приведшее к исключительной ситуации, возникает на другом компьютере или в другом процессе, для отслеживания таких ошибок необходим специальный метод-обработчик

```
TReconcileErrorEvent = procedure(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction)
of object;
property OnReconcileError: TReconcileErrorEvent;
```

который срабатывает, если с сервера пересылается сообщение об ошибке. Информация об ошибке содержится в параметре `E: EReconcileError`.

Более детальная информация о клиентских наборах данных содержится в *главе 29*.

Способы редактирования данных

Несмотря на декларированные недостатки технологии dbExpress — однонаправленные курсоры и невозможность редактирования — существуют программные способы уменьшить масштаб проблемы или даже решить ее.

Во-первых, в нашем распоряжении имеется компонент `TSimpleDataSet`, который реализует двунаправленный курсор и обеспечивает редактирование данных путем их кэширования на клиентской стороне.

Во-вторых, редактирование можно обеспечить настройкой и выполнением запросов SQL INSERT, UPDATE и DELETE.

У каждого способа есть свои преимущества и недостатки.

Компонент `TSimpleDataSet` безусловно хорош. Он технологичен, относительно прост в использовании и, главное, прячет всю функциональность за несколькими свойствами и методами. Но локальное кэширование изменений подходит далеко не для всех приложений.

Например, при многопользовательском интенсивном доступе к данным с их редактированием при локальном кэшировании могут возникнуть проблемы с целостностью и адекватностью данных. Самый распространенный пример: продавец в строительном супермаркете обслуживая покупателя в пик летних ремонтов резервирует несколько наименований ходовых товаров. Но покупатель замешкался, выбирая обои и плитку. А за это время другой продавец уже продал другому покупателю (заказ первого покупателя еще находится в локальном кэше!) часть его товаров.

Конечно, фиксацию изменений на сервере можно выполнять после локального сохранения каждой записи, но это приведет к загрузке соединения и снижению эффективности системы.

Применение модифицирующих запросов, с одной стороны, позволяет оперативно вносить изменения в данные на сервере, а с другой, — требует больших затрат на программирование и отладку. Сложность кода с этом случае существенно выше.

Рассмотрим примеры реализации обоих способов. В листинге 21.4 приложение Demo DBX использует соединение с сервером InterBase. Подключена тестовая база данных ..\Borland Shared\Data\MastSQL.gdb.

Листинг 21.4. Первый вариант приложения dbExpress с редактируемыми наборами данных

```
implementation
{$R *.dfm}
procedure TfmDemoDBX.FormCreate(Sender: TObject);
```

```
begin
 tblVens.Open;
 cdsCusts.Open;
end;
procedure TfmDemoDBX.FormDestroy(Sender: TObject);
begin
 tblVens.Close;
 cdsCusts.Close;
end;
{Editing feature with updating query}
procedure TfmDemoDBX.tblVensAfterScroll(DataSet: TDataSet);
begin
 edVenNo.Text := tblVens.FieldByName('VENDORNO').AsString;
 edVenName.Text := tblVens.FieldByName('VENDORNAME').AsString;
 edVenAdr.Text := tblVens.FieldByName('ADDRESS1').AsString;
 edVenCity.Text := tblVens.FieldByName('CITY').AsString;
 edVenPhone.Text := tblVens.FieldByName('PHONE').AsString;
end;
procedure TfmDemoDBX.sbCancelClick(Sender: TObject);
begin
 tblVens.First;
end;
procedure TfmDemoDBX.sbNextClick(Sender: TObject);
begin
 tblVens.Next;
end;
procedure TfmDemoDBX.sbPostClick(Sender: TObject);
begin
 with quUpdate do
 try
  ParamByName('Idx').AsInteger :=
  tblVens.FieldByName('VENDORNO').AsInteger;
  ParamByName('No').AsString := edVenNo.Text;
  ParamByName('Name').AsString := edVenName.Text;
  ParamByName('Adr').AsString :=  edVenAdr.Text;
  ParamByName('City').AsString :=  edVenCity.Text;
  ParamByName('Phone').AsString :=  edVenPhone.Text;
  ExecSQL;
 except
```

```
  MessageDlg('Vendor''s info post error', mtError, [mbOK], 0);
  tblVens.First;
 end;
end;
{Editing feature with cached updates}
procedure TfmDemoDBX.cdsCustsAfterPost(DataSet: TDataSet);
begin
 cdsCusts.ApplyUpdates(-1);
end;
procedure TfmDemoDBX.cdsCustsReconcileError(DataSet:
TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
begin
 MessageDlg('Customer''s info post error', mtError, [mbOK], 0);
 cdsCusts.CancelUpdates;
end;
end.
```

Для просмотра и редактирования выбраны таблицы VENDORS и CUSTOMERS. Первая таблица подключена через настроенное соединение (компонент `cnMast`) к компоненту `tblVens` типа `TSQLTable`. Значение пяти полей отображается в обычных компонентах `TEdit`, т. к. компоненты отображения данных, связанные с компонентом dbExpress через компонент `TDataSource`, работают только в режиме просмотра, не позволяя редактировать данные.

Благодаря методу-обработчику `AfterScroll` легко решилась проблема заполнения компонентов `TEdit` при навигации по набору данных. Для сохранения сделанных изменений (нажатие на кнопку `sbPost`) используется компонент `quUpdate` типа `TSQLQuery`. В параметрах запроса передаются текущие значения полей из компонентов `TEdit`. Так как в этом случае работает однонаправленный курсор, проблема обновления набора данных после выполнения модифицирующего запроса не возникает и набор данных обновляется только при вызове метода `First` компонента `tblVens`. На рис. 21.3 представлено окно приложения Demo DBX.

Вторая таблица подключена через тот же компонент `cnMast` к компоненту `cdsCusts` типа `TSimpleDataSet`. Он работает в табличном режиме. Данные отображаются в обычном компоненте `TDBGrid`.

Для сохранения сделанных изменений здесь задействован метод `ApplyUpdates`, размещенный в методе-обработчике `AfterPost`, когда изме-

нения уже попали в локальный кэш. Метод-обработчик вызывается каждый раз при переходе в компоненте `TDBGrid` на новую строку.

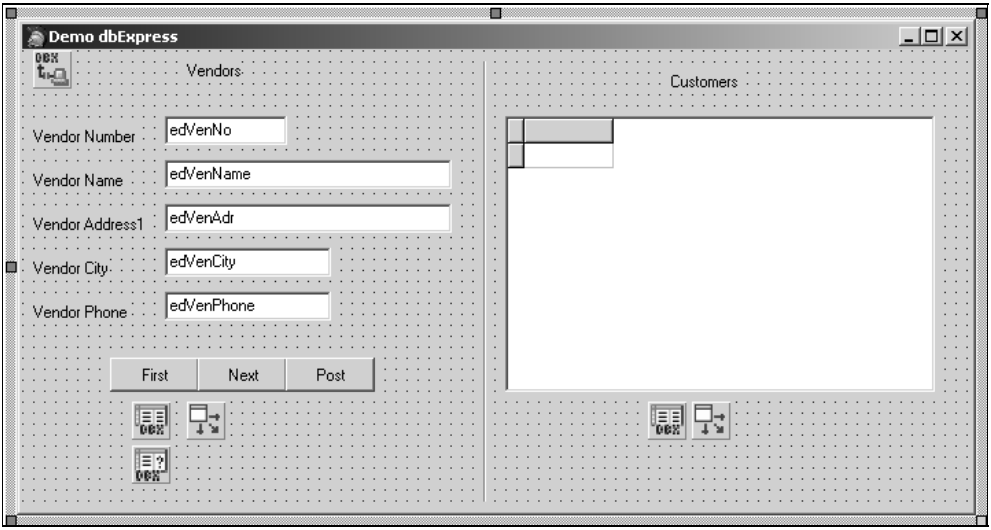


Рис. 21.3. Окно приложения Demo DBX

Для компонента `cdsCusts` также предусмотрена простейшая обработка исключительных ситуаций, возникающих на сервере.

Обратите также внимание на настройку компонента `cnMast` типа `TSQLConnection`. Свойства `KeepConnection` и `LoginPrompt` со значениями `False` обеспечивают открытие наборов данных при создании формы и автоматическое закрытие соединения при закрытии приложения с минимальным исходным кодом.

Как мы уже упоминали, редактирование можно обеспечить настройкой и выполнением запросов SQL INSERT, UPDATE и DELETE. Это позволяет оперативно вносить изменения в данные на сервере, но требует больших затрат на программирование и отладку.

Рассмотрим еще один пример (листинг 21.5). Здесь приложение Demo DBX также использует соединение с сервером InterBase. Подключена тестовая база данных ..\Program Files\Common Files\Borland Shared\Data\MastSQL.gdb.

Листинг 21.5. Второй вариант приложения dbExpress с редактируемыми наборами данных

```
implementation
{$R *.dfm}
```

```
procedure TfmDemoDBX.FormCreate(Sender: TObject);
begin
 tblVens.Open;
end;
procedure TfmDemoDBX.FormDestroy(Sender: TObject);
begin
 tblVens.Close;
end;
procedure TfmDemoDBX.tblVensAfterScroll(DataSet: TDataSet);
begin
 edVenNo.Text := tblVens.FieldByName('VENDORNO').AsString;
 edVenName.Text := tblVens.FieldByName('VENDORNAME').AsString;
 edVenAdr.Text := tblVens.FieldByName('ADDRESS1').AsString;
 edVenCity.Text := tblVens.FieldByName('CITY').AsString;
 edVenPhone.Text := tblVens.FieldByName('PHONE').AsString;
end;
procedure TfmDemoDBX.sbCancelClick(Sender: TObject);
begin
 tblVens.First;
end;
procedure TfmDemoDBX.sbNextClick(Sender: TObject);
begin
 tblVens.Next;
end;
procedure TfmDemoDBX.sbPostClick(Sender: TObject);
begin
 with quUpdate do
 try
  ParamByName('Idx').AsInteger :=
tblVens.FieldByName('VENDORNO').AsInteger;
  ParamByName('No').AsString := edVenNo.Text;
  ParamByName('Name').AsString := edVenName.Text;
  ParamByName('Adr').AsString :=  edVenAdr.Text;
  ParamByName('City').AsString :=  edVenCity.Text;
  ParamByName('Phone').AsString :=  edVenPhone.Text;
  ExecSQL;
 except
```

```
  MessageDlg('Vendor''s info post error', mtError, [mbOK], 0);
  tblVens.First;
 end;
end;
end.
```

Для просмотра и редактирования выбрана таблица VENDORS, которая подключена через настроенное соединение (компонент `cnMast`) к компоненту `tblVens` типа `TSQLTable`. Значение пяти полей отображается в обычных компонентах `TEdit`, т. к. компоненты отображения данных, связанные с компонентом dbExpress через компонент `TDataSource`, работают только в режиме просмотра, не позволяя редактировать данные.

Использование метода-обработчика `AfterScroll` позволило легко решить проблему заполнения компонентов `TEdit` при навигации по набору данных. Для сохранения сделанных изменений (нажатие на кнопку `sbPost`) используется компонент `quUpdate` типа `TSQLQuery`. В параметрах запроса передаются текущие значения полей из компонентов `TEdit`. Поскольку в этом случае работает однонаправленный курсор, проблема обновления набора данных после выполнения модифицирующего запроса не возникает и набор данных обновляется только при вызове метода `First` компонента `tblVens`.

Обратите также внимание на настройку компонента `cnMast` типа `TSQLConnection`. Свойства `KeepConnection` и `LoginPrompt` действуют так же, как и в предыдущем примере (см. листинг 21.4).

Интерфейсы dbExpress

Технология dbExpress основана на четырех базовых интерфейсах, методы которых существуют во всех компонентах dbExpress. При серьезной работе с технологией или проектировании собственных компонентов информация об этих интерфейсах будет полезна.

Интерфейс *ISQLDriver*

Интерфейс `ISQLDriver` инкапсулирует всего три метода для обслуживания драйвера dbExpress. Экземпляр интерфейса создается для соединения и обеспечивает его связь с драйвером.

Методы

```
function SetOption(eDOption: TSQLDriverOption; PropValue: LongInt):
SQLResult; stdcall;
```

```
function GetOption(eDOption: TSQLDriverOption; PropValue: Pointer;
MaxLength: SmallInt; out Length: SmallInt): SQLResult; stdcall;
```

позволяют работать с параметрами драйвера. А метод

```
function getSQLConnection(out pConn: ISQLConnection): SQLResult; stdcall;
```

возвращает указатель на интерфейс связанного с драйвером соединения `ISQLConnection`.

Получить доступ к интерфейсу `ISQLDriver` разработчик может, использовав защищенное свойство

```
property Driver: ISQLDriver read FSQLDriver;
```

компонента `TSQLConnection`.

Интерфейс *ISQLConnection*

Интерфейс `ISQLConnection` обеспечивает работу соединения. Он передает запросы серверу и возвращает результаты, создавая экземпляры интерфейса `ISQLCommand`; управляет транзакциями, поддерживает передачу метаданных при помощи интерфейса `ISQLMetaData`.

Для открытия соединения используется метод

```
function connect(ServerName: PChar; UserName: PChar; Password: PChar):
SQLResult; stdcall;
```

Здесь

- `ServerName` — имя базы данных,
- `UserName` — имя пользователя;
- `Password` — пароль пользователя.

Закрывает соединение метод

```
function disconnect: SQLResult; stdcall;
```

Параметры соединения управляются методами

```
function SetOption(eConnectOption: TSQLConnectionOption; lValue:
LongInt): SQLResult; stdcall;
function GetOption(eDOption: TSQLConnectionOption; PropValue: Pointer;
MaxLength: SmallInt; out Length: SmallInt): SQLResult; stdcall;
```

Для обработки запроса, проходящего через соединение, создается интерфейс `ISQLCommand`

```
function getSQLCommand(out pComm: ISQLCommand): SQLResult; stdcall;
```

Обработка транзакций осуществляется тремя методами:

```
function beginTransaction(TranID: LongWord): SQLResult; stdcall;
function commit(TranID: LongWord): SQLResult; stdcall;
function rollback(TranID: LongWord): SQLResult; stdcall;
```

А при помощи метода

```
function getErrorMessage(Error: PChar): SQLResult; overload; stdcall;
```

организована обработка исключительных ситуаций в компоненте `TSQLConnection`. В нем реализована защищенная процедура `SQLError`, которую можно применять в собственных компонентах и при необходимости дорабатывать.

Доступ к методам интерфейса `ISQLConnection` можно получить через пространство имен `Borland.Vcl.DBXpress`

Например, можно написать собственную процедуру контроля ошибок примерно по такому образцу (листинг 21.6).

Листинг 21.6. Пример процедуры контроля ошибок

```
procedure CheckError(IConn: ISQLConnection);
var FStatus: SQLResult;
    FSize:SmallInt;
    FMessage: pChar;
begin
 FStatus := IConn.getErrorMessageLen(FSize);
 if (FStatus = SQL_SUCCESS)and(FSize > 0) then
 begin
  FMessage := AllocMem(FSize + 1);
  FStatus := IConn.getErrorMessage(FMessage);
  if FStatus = SQL_SUCCESS
   then MessageDlg(FMessage, mtError, [mbOK], 0)
   else MessageDlg('Checking error', mtWarning, [mbOK], 0);
  if Assigned(FMessage)
   then FreeMem(FMessage);
 end;
end;
```

Доступ к интерфейсу `ISQLConnection` можно получить через свойство

```
property SQLConnection: ISQLConnection;
```

компонента `TSQLConnection`.

Интерфейс *ISQLCommand*

Интерфейс `ISQLCommand` обеспечивает функционирование запроса dbExpress. Компоненты dbExpress, работающие с наборами данных, реализуют через него свои методы.

Параметры запроса устанавливаются методом

```
function setParameter(ulParameter: Word ; ulChildPos: Word ; eParamType:
TSTMTParamType ; uLogType: Word; uSubType: Word; iPrecision: Integer;
iScale: Integer; Length: LongWord ; pBuffer: Pointer; lInd: Integer):
SQLResult; stdcall;
```

Здесь

- `uParameterNumber` — порядковый номер параметра;
- `uChildPos` — задает порядковый номер параметра, если он является дочерним для сложных типов данных;
- `ePType` — задает тип параметра (входной, выходной смешанный);
- `uLogType` — тип данных параметра;
- `uSubType` — вспомогательный параметр типа данных;
- `lMaxPrecision` — максимальный размер значения в байтах;
- `lMaxScale` — максимальная точность типа данных;
- `ulLength` — размер буфера;
- `pBuffer` — буфер, содержащий значение параметра;
- `bIsNull` — флаг, определяющий, может ли параметр иметь нулевое значение.

Для каждого параметра метод вызывается снова.

Информацию о параметре можно получить методом

```
function ISQLCommand_getParameter(Handle: ISQLCommand; ParameterNumber:
Word; ulChildPos: Word; Value: IntPtr; Length: Integer; var IsBlank:
Integer): SQLResult;
```

Здесь

- `ParameterNumber` — порядковый номер параметра;
- `ulChildPos` — задает порядковый номер параметра, если он является дочерним для сложных типов данных;
- `Value` — указатель на буфер значения параметра;
- `Length` — размер буфера;
- `IsBlank` — признак незаполненного параметра.

Метод

```
function Prepare(SQL: PChar; ParamCount: Word): SQLResult; stdcall;
```

готовит запрос к выполнению с учетом значений параметров.

Выполнение запроса осуществляется методом

```
function Execute(var Cursor: ISQLCursor): SQLResult; stdcall;
```

который возвращает в параметре интерфейс курсора, если запрос выполнен,

или методом

```
function ExecuteImmediate(SQL: PChar; var Cursor: ISQLCursor): SQLResult;
stdcall;
```

который выполняет запрос, не требующий подготовки (не имеющий параметров). Он также возвращает в параметре `Cursor` готовый интерфейс курсора, если запрос выполнен успешно. Текст запроса определяется параметром SQL.

Метод

```
function getNextCursor(var Cursor: ISQLCursor): SQLResult; stdcall;
```

определяет в параметре `Cursor` курсор следующего набора данных, если выполнялась хранимая процедура, и она возвращает несколько наборов данных.

Интерфейс `ISQLCommand` используется компонентом `TCustomSQLDataSet` и недоступен потомкам.

Интерфейс *ISQLCursor*

Интерфейс `ISQLCursor` обладает совокупностью методов, которые помогут получить информацию о полях курсора, а также значения этих полей. Все эти методы имеют одинаковое представление. Для получения нужной информации необходимо задать порядковый номер поля в структуре курсора.

Метод

```
function Next: SQLResult; stdcall;
```

обновляет курсор, занося в него информацию из следующей строки набора данных.

Интерфейс `ISQLCursor` используется компонентом `TCustomSQLDataSet` и недоступен потомкам.

Отладка приложений с технологией dbExpress

Наряду с обычными методами отладки исходного кода, в dbExpress существует возможность контроля запросов, проходящих на сервер через соединение. Для этого применяется компонент `TSQLMonitor`.

Через свойство

```
property SQLConnection: TSQLConnection;
```

компонент связывается с отлаживаемым соединением.

Затем компонент включается установкой `Active = True`.

Теперь во время выполнения приложения сразу после открытия соединения свойство

```
property TraceList: TStrings;
```

будет заполняться информацией обо всех проходящих командах.

Содержимое этого списка можно сохранить в файле при помощи метода

```
procedure SaveToFile(AFileName: string);
```

Эту же информацию можно автоматически добавлять в текстовый файл, определяемый свойством

```
property FileName: string;
```

но только тогда, когда свойство

```
property AutoSave: Boolean;
```

будет иметь значение `True`.

Свойство

```
property MaxTraceCount: Integer;
```

определяет максимальное число контролируемых команд, а также управляет процессом контроля. При значении −1 ограничения снимаются, а при значении 0 контроль останавливается.

Текущее число проверенных команд содержится в свойстве

```
property TraceCount: Integer;
```

Перед записью команды в список вызывается метод-обработчик

```
TTraceEvent = procedure(Sender: TObject; CBInfo: pSQLTRACEDesc; var
LogTrace: Boolean) of object;
property OnTrace: TTraceEvent;
```

а сразу поле записи в список вызывается

```
TTraceLogEvent = procedure (Sender: TObject; CBInfo: pSQLTRACEDesc) of
object;
property OnLogTrace: TTraceLogEvent;
```

Таким образом, разработчик получает компактный и удобный компонент, позволяющий без усилий получать информацию о прохождении команд в соединении.

Если же компонент `TSQLMonitor` не подходит, можно воспользоваться методом

```
procedure SetTraceCallbackEvent(Event: TSQLCallbackEvent; IClientInfo:
Integer);
```

компонента `TSQLConnection`. Параметр процедурного типа `Event` определяет функцию, которая будет вызываться при выполнении каждой команды. Параметр `IClientInfo` должен содержать любое число.

Он позволяет разработчику самостоятельно определить функцию типа `TSQLCallbackEvent`:

```
TRACECat = TypedEnum;
TSQLCallbackEvent = function(CallType: TRACECat; CBInfo: Pointer):
CBRType; stdcall;
```

Эта функция будет вызываться каждый раз при прохождении команды. Текст команды будет передаваться в буфер `CBInfo`. Разработчику необходимо лишь выполнить запланированные действия с буфером внутри функции.

Рассмотрим в качестве примера следующий исходный код (листинг 21.7).

Листинг 21.7. Пример отладки приложения

```
function GetTraceInfo(CallType: TRACECat; CBInfo: Pointer): CBRType;
stdcall;
begin
 if Assigned(Form1.TraceList) then Form1.TraceList.Add(pChar(CBinfo));

end;
procedure TForm1.MyConnectionBeforeConnect(Sender: TObject);
begin
 TraceList := TStringList.Create;
end;
procedure TForm1.MyConnectionAfterDisconnect(Sender: TObject);
begin
 if Assigned(TraceList) then
 begin
  TraceList.SaveToFile('c:\Temp\TraceInfo.txt');
  TraceList.Free;
 end;
end;
procedure TForm1.StartBtnClick(Sender: TObject);
begin
  MyConnection.SetTraceCallbackEvent(GetTraceInfo, 8);
  MyConnection.Open;
  {...}
  MyConnection.Close;
end;
```

Перед открытием соединения в методе-обработчике `BeforeConnection` создается объект типа `TStringList`. После закрытия соединения этот объект сохраняется в файле и уничтожается.

Перед открытием соединения (метод-обработчик нажатия кнопки **Start**) при помощи метода `SetTraceCallbackEvent` с соединением связывается функция `GetTraceInfo`.

Таким образом, по мере прохождения команд информация о них будет накапливаться в списке. После закрытия соединения список сохраняется в текстовом файле.

Примечание

В своей работе компонент `TSQLMonitor` также осуществляет вызовы метода `SetTraceCallbackEvent`. Поэтому одновременно использовать компонент и собственные функции нельзя.

Распространение приложений с технологией dbExpress

Готовое приложение, использующее технологию dbExpress, можно поставлять заказчикам двумя способами.

Вместе с приложением поставляется динамическая библиотека для выбранного сервера (см. табл. 21.1). Она находится в папке ..\Bin.

Дополнительно, если в приложении встречается компонент `TSimpleDataSet`, необходимо включить в поставку динамическую библиотеку midas.dll.

Приложение компилируется вместе со следующими DCU-файлами: dbExpInt.dcu, dbExpOra.dcu, dbExpDb2.dcu, dbExpMy.dcu и т. д. (в зависимости от выбранного сервера). Если в приложении присутствует компонент `TSimpleDataSet`, необходимо добавить файлы Crtl.dcu и MidasLib.dcu. В результате необходимо поставлять только исполняемый файл приложения.

Если дополнительная настройка соединений не требуется, файл dbxconnections.ini не нужен.

Резюме

Технология dbExpress предназначена для создания приложений, требующих быстрого доступа к базам данных, хранящимся на серверах SQL. Доступ осуществляется при помощи небольших драйверов, реализованных в виде динамических библиотек.

В настоящее время созданы драйверы для следующих серверов баз данных:

- DB2
- Informix
- InterBase
- MS SQL Server
- MySQL
- Oracle
- Sybase Adaptive Server Anywhere (ASA)
- Sybase Adaptive Server Enterprise (ASE)

Технология dbExpress реализована на основе стандартных типов компонентов доступа к данным, проста при распространении (исполняемый файл приложения или одна-две динамические библиотеки), поддерживает кросс-платформенную разработку для Linux и легко интегрируется в приложения CLX.

Недостатки технологии — однонаправленные курсоры и ограниченные возможности по редактированию (редактирование возможно только при кэшировании изменений на клиенте или выполнением специальных модифицирующих запросов).

Глава 22

Технология InterBase Express

В категории **InterBase** Палитры инструментов содержатся компоненты доступа к данным, адаптированные для работы с сервером InterBase, и объединенные названием **InterBase Express**. Компоненты из набора **InterBase Express** предназначены для работы с сервером InterBase версии не ниже 5.5.

Преимущество заключается в реализации всех функций за счет прямого обращения к API сервера InterBase. Благодаря этому существенно повысилась скорость работы компонентов.

Новые компоненты предоставляют разработчику дополнительные возможности. Среди них:

- улучшенное управление транзакциями (для этого теперь предусмотрен отдельный компонент `TIBTransaction`);
- новые компоненты доступа к данным, позволяющие лучше решать традиционные задачи программирования (компоненты `TIBDataSet`, `TIBSQL`);
- возможность получения сведений о состоянии базы данных без прямого обращения к ее системным таблицам (компонент `TIBDatabaseInfo`);
- отслеживание состояния процессов выполнения запросов (компонент `TIBSQLMonitor`).

С точки зрения разработчика, за исключением нескольких новых свойств, методика работы с этими компонентами в приложениях БД не отличается от стандартной *(см. главу 15)*. Любой новый компонент, инкапсулирующий набор данных, совершенно обычным образом через компонент `TDataSource` можно подключить к любому стандартному компоненту отображения данных.

В этой главе рассматриваются следующие вопросы.

- Соединение с сервером InterBase и полноценное управление транзакциями из клиентского приложения.
- Изменения в стандартных компонентах доступа к данным.
- Назначение и возможности новых компонентов доступа к данным.

- Отслеживание процессов выполнения запросов на сервере из клиентского приложения.
- Оценка состояния базы данных.
- Особенности переноса клиентских приложений, работающих с сервером InterBase, на новую компонентную базу.

Механизм доступа к данным InterBase Express

Для компонентов InterBase Express соединение с сервером БД осуществляет компонент `TIBDatabase`.

Для создания клиент-серверного приложения необходимо не только иметь работающий сервер, но и инсталлировать на клиентских рабочих местах специальное программное обеспечение, выполняющее соединение клиентского приложения с сервером.

Механизм доступа к данным InterBase Express использует для обращений к серверу возможности клиентского ПО InterBase, которое должно быть инсталлировано на компьютере. Если с данного компьютера доступны базы данных какого-либо сервера на платформе InterBase, то рассматриваемые здесь компоненты могут обращаться к этому серверу. При этом не требуется BDE или какой-либо другой механизм доступа к данным.

Но в результате все компоненты InterBase Express, инкапсулирующие набор данных, должны обращаться к базе данных только через компонент соединения `TIBDatabase`. На самом деле эта особенность не является недостатком в клиентских приложениях, т. к. организация соединения через один специализированный компонент всячески приветствуется и является хорошим тоном в программировании.

Компонент *TIBDatabase*

Так как для доступа к базе данных компонентам InterBase Express не требуется BDE, то для создания соединения достаточно всего одного свойства `DatabaseName`. В нем необходимо указать полный путь (включая имя сервера) к выбранному файлу БД с расширением GDB. При щелчке на кнопке свойства в Инспекторе объектов для этого можно воспользоваться стандартным диалогом выбора файла.

Компонент имеет собственный редактор, который позволяет задать значения основных свойств, обеспечивающих соединение с базой данных (рис. 22.1).

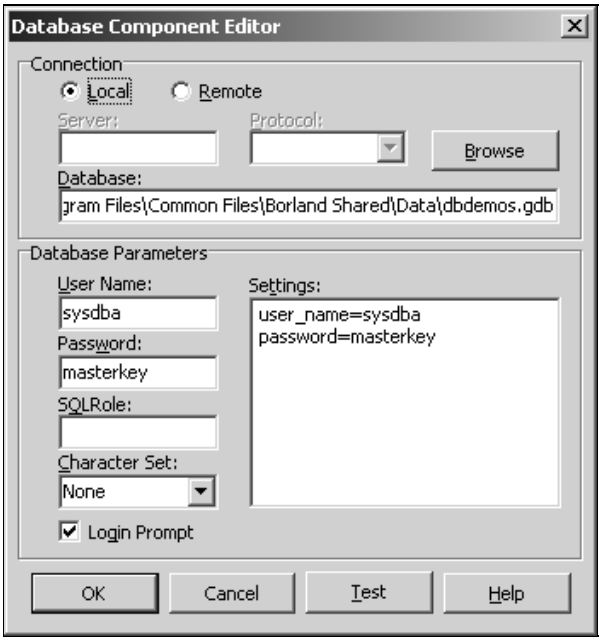


Рис. 22.1. Редактор компонента `TIBDatabase`

Настройка соединения проводится следующим образом.

1. В панели **Connection** выбирается требуемый сервер InterBase (локальный или доступный удаленно).
2. При помощи списка **Protocol** определяется используемый сетевой протокол.
3. Кнопкой **Browse** выбирается файл базы данных.
4. В панели **Database Parameters** задаются имя пользователя, его пароль и роль. Также можно выбрать и набор шрифтов для языковой адаптации приложения (список **Character Set**).

Задать вводимые при подключении параметры (имя пользователя, пароль, схему, роль и т. д.) можно также с помощью свойств `Params` и `LoginPrompt`.

Путь к файлу базы данных задается свойством

```
property DatabaseName: String;
```

Соединение включается и отключается свойством

```
property Connected : Boolean;
```

При этом свойство

```
property AllowStreamedConnected : Boolean;
```

управляет включением соединения при запуске приложения и служит дополнительным предохранителем. При значении `False` свойство запрещает открытие соединения при запуске приложения, даже если свойство `Connected` имело значение `True`. Поскольку приложение часто отлаживается на тестовой базе данных, а используется совсем с другими, то неверный путь в свойстве `DatabaseName` и не отключенное на этапе разработки свойство `Connected` приведут к возникновению ошибки открытия соединения при запуске приложения на другом компьютере.

Параметры соединения, которые нельзя задать свойствами, устанавливаются свойством

```
property Params: TStrings;
```

в котором в каждой строке задается имя параметра и затем через знак равенства — его значение. Наиболее распространенный пример применения свойства `Params` — задание имени пользователя и его пароля:

```
user_name=sysdba
password=masterkey
```

Свойство

```
property DBParamByDPB: [const Idx: Integer]: String;
```

позволяет получить доступ к отдельным параметрам соединения, не обращаясь к свойству `Params`.

Примечание

Полный список индексов всех возможных параметров соединения InterBase можно найти в файле ..\Source\Win32\IBX\IBHeader.pas.

Если соединение настроено правильно, метод

```
procedure TestConnected: Boolean;
```

возвращает `True`, иначе — `False`.

Свойство

```
property IdleTimer: Integer;
```

задает временной интервал до отключения неиспользуемого соединения.

В компоненте `TIBDatabase` отсутствуют средства управления транзакциями, которые вынесены в отдельный компонент `TIBTransaction`.

Свойство

```
property DefaultTransaction: TIBTransaction;
```

позволяет задать транзакцию по умолчанию. При этом все компоненты с наборами данных, работающие с данным соединением, автоматически на-

чинают применять этот компонент транзакции. Изменяя значение этого свойства можно в одном соединении работать с несколькими транзакциями.

Общее число связанных с данным соединением транзакций возвращает свойство

```
property TransactionCount: Integer;
```

а их полный перечень содержится в индексированном списке свойства

```
property Transactions [Index: Integer]: TIBTransaction;
```

Добавить новую транзакцию к списку можно при помощи метода

```
function AddTransaction(TR: TIBTransaction): Integer;
```

Отменить связь между соединением и компонентом транзакции позволяет метод

```
procedure RemoveTransaction(Idx: Integer);
```

Но можно поступить и более радикально. Метод

```
procedure RemoveTransactions;
```

отменяет связи со всеми транзакциями.

Используемый в методе `RemoveTransaction` индекс транзакции может быть найден методом

```
function FindTransaction (TR: TIBTransaction): Integer;
```

а метод

```
function FindDefaultTransaction: TIBTransaction;
```

возвращает транзакцию по умолчанию.

С компонентом соединения можно связать произвольное число объектов, отслеживающих возникновение событий в базе данных InterBase. Для этого предусмотрен метод

```
procedure AddEventNotifier(Notifier: IIBEventNotifier);
```

который связывает с соединением либо интерфейс `IIBEventNotifier`, либо объект `TIBEvents`.

Парный ему метод

```
procedure RemoveEventNotifier(Notifier: IIBEventNotifier);
```

разрывает связь соединения с объектом-обработчиком событий.

Свойство

```
type
TTraceFlag = (tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt,
tfConnect, tfTransact, tfBlob, tfService, tfMisc);
```

```
TTraceFlags = set of TTraceFlag;
property TraceFlags: TTraceFlags;
```

позволяет управлять сведениями о выполнении запросов, возвращаемыми компонентом `TSQLMonitor` (см. далее описание этого компонента).

Группа методов позволяет судить о реальном состоянии соединения во время выполнения. Все они в случае неудачи проверки генерируют исключение `EIBClientError`.

Методы

```
procedure CheckActive;
```

и

```
procedure CheckInactive;
```

проверяют, функционирует соединение или нет.

Метод

```
procedure CheckDatabaseName;
```

проверяет, заполнено ли свойство `DatabaseName`.

Компонент `TIBDatabase` позволяет выполнять некоторые операции с метаданными базы данных.

При помощи метода

```
procedure CreateDatabase;
```

можно создавать новые базы данных, включая создание файла базы данных. Все параметры новой базы данных, которые разработчик посчитает нужным указать явно, должны быть включены в список свойства `Params`.

Имя файла новой базы данных должно быть указано в свойстве `DatabaseName`.

Метод

```
procedure DropDatabase;
```

удаляет существующую базу данных, путь к которой указан свойством `DatabaseName`.

Список `List` имен таблиц, имеющихся в базе данных, возвращает метод

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean =
False);
```

При этом параметр `SystemTables` управляет включением в список имен системных таблиц.

А метод

```
procedure GetFieldNames(const TableName: string; List: TStrings);
```

аналогичным образом возвращает список полей для таблицы, заданной параметром `TableName`.

Методы-обработчики событий компонента `TIBDatabase` представлены в табл. 22.1.

Таблица 22.1. *Методы-обработчики событий компонента* `TIBDatabase`

Объявление	Тип	Описание
`property AfterConnect: TNotifyEvent;`	Pb	Выполняется после открытия соединения
`property AfterDisconnect: TNotifyEvent;`	Pb	Выполняется после закрытия соединения
`property BeforeConnect: TNotifyEvent;`	Pb	Выполняется перед открытием соединения
`property BeforeDisconnect: TNotifyEvent;`	Pb	Выполняется перед закрытием соединения
`property OnDialectDowngradeWarning: TNotifyEvent;`	Pb	Выполняется в случае изменения диалекта SQL при открытии соединения
`property OnIdleTimer: TNotifyEvent;`	Pb	Вызывается при истечении времени, заданного свойством `IdleTimer`
`TDatabaseLoginEvent = procedure(Database: TIBDatabase; LoginParams: TStrings) of object;` `property OnLogin: TDatabaseLoginEvent;`	Pb	Вызывается для регистрации пользователя при открытии соединения

Компонент *TIBTransaction*

Компонент `TIBTransaction` инкапсулирует средства управления транзакцией при работе с сервером InterBase. Для этого он должен быть связан с компонентом `TIBDatabase` при помощи своего свойства

```
property DefaultDatabase: TIBDatabase;
```

Один компонент транзакции может быть связан с несколькими компонентами `TIBDatabase`. Для этого необходимо задать один компонент транзакции в свойствах `DefaultTransaction` всех необходимых компонентов соединений.

Список всех связанных компонентов соединений содержится в свойстве

```
property Databases[Index: Integer]: TIBDatabase;
```

а их общее число возвращает свойство

```
property DatabaseCount: Integer;
```

Во время выполнения новое соединение может быть связано с транзакцией методом

```
function AddDatabase(db: TIBDatabase): Integer;
```

Или же, связь может быть отменена:

```
procedure RemoveDatabase(Idx: Integer);
```

А метод

```
procedure RemoveDatabases;
```

разрывает все установленные связи с компонентами `TIBDatabase`.

Индекс связанного соединения в списке `Databases` транзакции можно получить при помощи метода

```
function FindDatabase (db: TIBDatabase): Integer;
```

Например, если вам не известно ничего, кроме имени компонента, можно поступить так:

```
var i, FIndex: Integer;
...
for i := 0 to Form1.ComponentCount – 1 do
  if Form1.Components[i].Name = 'IBDatabase1'
   then FIndex := IBTransac-
tion1.FindDatabase(TIBDatabase(Form1.Components[i]));
...
```

Соединение, заданное по умолчанию свойством `DefaultDatabase`, возвращает метод

```
function FindDefaultDatabase: TIBDatabase;
```

Транзакция может иметь набор параметров, задать которые можно при помощи свойства

```
property Params: TStrings;
```

аналогично компоненту `TIBDatabase`. А прямой доступ для чтения к буферу параметров транзакции Transaction Parameters Buffer (TPB) типа `PChar` обеспечивает свойство

```
property TPB: PChar;
```

Длина буфера содержится в свойстве

```
property TPBLength: Short;
```

Дескриптор транзакции представлен свойством

```
property Handle: TISC_TR_HANDLE;
```

После того как транзакция настроена, ее можно начать, сохранить или отменить.

Транзакция стартует при помощи метода

```
procedure StartTransaction;
```

При необходимости сохранить все сделанные в рамках текущей транзакции изменения используется метод

```
procedure Commit;
```

А если выполненные действия нужно отменить, применяется метод

```
procedure Rollback;
```

Открыть и сохранить транзакцию можно традиционным свойством

```
property Active: Boolean;
```

После начала новой транзакции свойство

```
property InTransaction: Boolean;
```

принимает значение `True`, а после фиксации или отката — значение `False`.

При работе с сервером InterBase, начиная с версии 6.0, допускаются методы `CommitRetaining` и `RollbackRetaining`. В отличие от стандартных операций фиксации и отката транзакций эти методы после передачи или отмены изменений оставляют текущую транзакцию открытой.

Если сервер перегружен и не откликается на транзакцию, то после истечения времени, заданного свойством

```
property IdleTimer: Integer;
```

выполняется действие, заданное свойством

```
type TTransactionAction = (taRollback, taCommit,
taRollbackRetaining, taCommitRetaining);
property DefaultAction: TTransactionAction;
```

Здесь

- `taRollback` — откат транзакции;
- `taCommit` — фиксация транзакции;
- `taRollbackRetaining` — отмена изменений без завершения транзакции (для сервера InterBase 6.0);
- `taCommitRetaining` — фиксация изменений без завершения транзакции (для сервера InterBase 6.0).

Для компонента транзакции можно настроить ее автоматическое завершение при закрытии последнего открытого компонента, инкапсулирующего набор данных, связанного с тем же соединением, что и транзакция.

Для этого свойство

```
type TAutoStopAction = (saNone, saRollback, saCommit,
saRollbackRetaining, saCommitRetaining);
property AutoStopAction : TAutoStopAction;
```

не должно иметь значение `saNone`.

Остальные значения свойства выполняют следующие действия:

- `saRollback` — откат транзакции;
- `saCommit` — фиксация транзакции;
- `saRollbackRetaining` — отмена изменений без завершения транзакции (начиная с сервера InterBase 6.0);
- `saCommitRetaining` — фиксация изменений без завершения транзакции (начиная с сервера InterBase 6.0).

А метод

```
procedure CheckAutoStop;
```

выполняет действие, предусмотренное текущим значением свойства `AutoStopAction`.

Диагностика состояния транзакции во время выполнения осуществляется группой специальных методов. В случае отрицательного результата все они генерируют исключение `EIBClientError`.

Метод

```
procedure CheckDatabasesInList;
```

проверяет, имеются ли в списке `Databases` связанные соединения.

Метод

```
procedure CheckInTransaction;
```

проверяет, открыта ли в данный момент транзакция.

Метод

```
procedure CheckNotInTransaction;
```

проверяет, закрыта ли в данный момент транзакция.

Единственный метод-обработчик транзакции

```
property OnIdleTimer: TNotifyEvent;
```

вызывается после истечения срока ожидания выполнения транзакции, заданного свойством `IdleTimer`

Компоненты доступа к данным

Так как компоненты InterBase Express используют для получения набора данных собственный механизм, то иерархия классов-предков включает только обязательный для всех наборов данных `TDataSet` и класс `TIBCustomDataSet`, который собственно и инкапсулирует механизм доступа InterBase Express.

Связь с базой данных компоненты InterBase Express реализуют через компоненты соединения `TIBDatabase`. Для этого есть свойство

```
property Database: TIBDatabase;
```

Доступ к связанной транзакции осуществляется через свойство

```
property Transaction: TIBTransaction;
```

Дополнительно к стандартным свойствам и методам, описываемым в *главе 16*, класс `TIBCustomDataSet` имеет свойство

```
type TIBUpdateRecordTypes = set of (cusModified, cusInserted,
cusDeleted, cusUnmodified, cusUninserted);
property UpdateRecordTypes: TIBUpdateRecordTypes;
```

определяющее записи набора данных, на которые распространяются операции кэширования.

Здесь

- `cusModified` — модифицированные записи;
- `cusInserted` — добавленные записи;
- `cusDeleted` — удаленные записи;
- `cusUnmodified` — немодифицированные записи;
- `cusUninserted` — недобавленные записи.

Свойство

```
property BufferChunks: Integer;
```

определяет число записей, которые компонент загружает в собственный локальный буфер для ускорения выполнения стандартных операций.

При использовании компонентов в приложениях необходимо учитывать некоторые особенности.

Обновление набора данных выполняется не при каждом сохранении изменений. Такое поведение компонента определяется свойством

```
property ForcedRefresh: Boolean;
```

которое по умолчанию имеет значение `False`. Это ускоряет работу компонента. При необходимости выполнять обновление данных с максимальной частотой свойству `ForcedRefresh` необходимо присвоить значение `True`.

В зависимости от настроек компонента с ним можно выполнять различные виды операций редактирования, перечень которых содержится в свойстве только для чтения

```
type
  TLiveMode = (lmInsert, lmModify, lmDelete, lmRefresh);
  TLiveModes = set of TLiveMode;
property LiveMode: TLiveModes;
```

Так как эти компоненты предназначены для работы с сервером, то изначально все они поддерживают режим кэширования изменений и имеют соответствующие свойства, методы и методы-обработчики событий (табл. 22.2).

Таблица 22.2. *Методы-обработчики событий класса* `TIBCustomDataSet`

Объявление	Описание
`property AfterDatabaseDisconnect: TNotifyEvent;`	Выполняется после закрытия соединения с базой данных
`property AfterTransactionEnd: TNotifyEvent;`	Выполняется после окончания транзакции, с которой связан данный набор данных
`property BeforeDatabaseDisconnect: TNotifyEvent;`	Выполняется перед закрытием соединения с базой данных
`property BeforeTransactionEnd: TNotifyEvent;`	Выполняется перед окончанием транзакции, с которой связан данный набор данных
`property DatabaseFree: TNotifyEvent;`	Выполняется при обнулении свойства `Database` компонента набора данных

`type`
`TIBUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied, uaApply);`
`TIBUpdateErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError; UpdateKind: TUpdateKind; var UpdateAction: TIBUpdateAction)of object;`
`property OnUpdateError: TIBUpdateErrorEvent;`
	Вызывается при возникновении ошибки сохранения изменений в режиме кэширования

Таблица 22.2 *(окончание)*

Объявление	Описание

`type`
`TIBUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApply, uaApplied);`
`TIBUpdateRecordEvent = procedure(DataSet: TDataSet; UpdateKind: TUpdateKind; var UpdateAction: TIBUpdateAction) of object;`
`property OnUpdateRecord: TIBUpdateRecordEvent;`
	Вызывается при сохранении изменений в режиме кэширования

`property TransactionFree: TNotifyEvent;`	Выполняется при обнулении свойства `Transaction` компонента набора данных

Возможности компонентов `TIBTable`, `TIBQuery`, `TIBStoredProc` и `TIBUpdateSQL` мало чем отличаются от стандартных, описанных в *главе 16*.

Для взаимодействия с сервером компоненты InterBase Express используют два класса, которые инкапсулируют важные структуры API InterBase. Эти структуры обеспечивают передачу серверу параметров запроса и возвращение результата выполнения запроса. Поэтому сначала рассмотрим классы `TIBXSQLDA` и `TIBXSQLVAR`, а затем перейдем к компонентам.

Область дескрипторов *XSQLDA*

Запрос может иметь собственные параметры, которые должны содержаться в свойстве `Params`. Однако, в отличие от обычного компонента запроса, в InterBase Express это свойство представляет собой экземпляр класса `TIBXSQLDA`. Этот класс инкапсулирует одноименную структуру API InterBase — `XSQLDA`, обеспечивающую передачу параметров запросу и возврат результатов. Такая структура имеется у каждого запроса, выполняющегося сервером InterBase, и называется областью дескрипторов запроса (descriptors area). Свойства и методы компонента `TIBXSQLDA` приведены в табл. 22.3.

Таблица 22.3. *Свойства и методы компонента* `TIBXSQLDA`

Объявление	Тип	Описание
Свойства		
`property AsXSQLDA: PXSQLDA;`	Pu	Ссылка на структуру `XSQLDA`
`property Count: Integer;`	Pu	Возвращает число полей в структуре
`property Modified: Boolean;`	Pu	Позволяет определить возможность редактирования полей структуры
`property Names: String;`	Pu	Возвращает имена полей в структуре
`property RecordSize: Integer;`	Pu	Возвращает размер записи структуры
`property Vars: [Idx: Integer]: TIBXSQLVAR;`	Pu	Индексированный список структур `XSQLVAR` (см. далее)
Методы		
`procedure AddName(FieldName: String; Idx: Integer);`	Pu	Добавляет к структуре новое поле
`function ByName: [Idx: String]: TIBXSQLVAR;`	Pu	Возвращает структуру `XSQLVAR`, инкапсулирующую отдельное поле результата запроса (см. далее)

Структура *XSQLVAR*

Рассмотренная ранее область дескрипторов содержит возвращаемый результат запроса. Массив значений каждого возвращаемого поля сохраняется в отдельной структуре `XSQLVAR`. Индексированный список таких структур в области дескрипторов представлен свойством

```
property Vars: [Idx: Integer]: TIBXSQLVAR
```

В целом, рассматриваемая структура соответствует объекту поля, о чем свидетельствует и набор основных свойств и методов класса структуры, представленный в табл. 22.4.

Помимо представленных в таблице, класс `TIBXSQLVAR` имеет ряд свойств, возвращающих значение в определенном формате: `AsCurrency`, `AsDate`, `AsDateTime`, `AsDouble`, `AsFloat`, `AsInt64`, `AsInteger`, `AsLong`, `AsPointer`, `AsQuad`, `AsShort`, `AsString`, `AsTime`, `AsVariant`.

Таблица 22.4. *Свойства и методы компонента* `TIBXSQLVAR`

Объявление	Тип	Описание
Свойства		
`property AsXSQLVAR: PXSQLVAR;`	Pu	Представляет значение поля в виде структуры `XSQLVAR`
`property Data: PXSQLVAR;`	Pu	Ссылка на структуру `XSQLVAR`
`property Index: Integer;`	Pu	Возвращает индекс структуры в области дескрипторов
`property IsNull: Boolean;`	Pu	Позволяет определить наличие данных в структуре
`property IsNullable: Boolean;`	Pu	Позволяет определить, может ли структура иметь значение
`property Modified: Boolean;`	Pu	Позволяет определить, изменялось ли значение в структуре
`property Size: Integer;`	Pu	Максимальный размер данных в байтах
`property SQLType: Integer;`	Pu	Возвращает индекс API параметра
`property Value: Variant;`	Pu	Содержит возвращаемое значение
Методы		
`procedure Assign(Source: TIBXSQLVAR);`	Pu	Присваивает объект, передаваемый в параметре данному объекту
`procedure LoadFrom-File(const FileName: String);`	Pu	Загружает из файла данные в поле BLOB
`procedure LoadFromStream(Stream: TStream);`	Pu	Загружает из потока данные в поле BLOB
`procedure Save-ToFile(const FileName: String);`	Pu	Сохраняет в файле данные из поля BLOB
`procedure SaveToStream(Stream: TStream);`	Pu	Сохраняет в потоке данные из поля BLOB

Компонент *TIBTable*

Компонент `TIBTable` реализует все возможности стандартного компонента, инкапсулирующего таблицу *(см. главу 16)*. Дополнительно к ним можно обратить внимание на несколько полезных свойств и методов.

При выборе таблицы (свойство `TableName`) свойство

```
type
  TIBTableType = (ttSystem, ttView);
  TIBTableTypes = set of TIBTableType;
property TableTypes: TIBTableTypes;
```

определяет, какие таблицы доступны для выбора:

- `ttSystem` — доступны системные таблицы и просмотры;
- `ttView` — доступны определенные пользователем просмотры.

При открытии набора данных упорядочивание записей осуществляется в соответствии со значением свойства

```
property DefaultIndex: Boolean;
```

При значении `True` записи располагаются в порядке, определяемом первичным индексом таблицы БД.

Во время выполнения свойство

```
property Exists: Boolean;
```

позволяет определить, существует ли в базе данных таблица, имя которой задано свойством `TableName`.

Метод

```
procedure GotoCurrent(Table: TIBTable);
```

синхронизирует курсоры текущего набора данных и набора данных компонента, заданного параметром `Table`.

Методы-обработчики событий полностью соответствуют классу `TIBCustomDataSet` (см. табл. 22.2).

Компонент *TIBQuery*

Компонент `TIBQuery` выполняет все стандартные функции компонента запроса и наследует возможности класса `TIBCustomDataSet`.

Как и в остальных компонентах запросов, свойство

```
property SQL: TStrings;
```

содержит текст запроса и позволяет редактировать его. С этим свойством связан специализированный редактор (рис. 22.2).

Для просмотра текста запроса можно использовать свойство

```
property Text: string;
```

Параметры запроса хранятся в стандартном свойстве

```
property Params: TParams;
```

а общее число параметров запроса возвращает свойство

```
property ParamCount: Word;
```

При создании новых записей в редактируемых наборах данных компонентов запросов возникает проблема присвоения значений полям первичных индексов. Очевидно, что при сохранении новой записи в базе данных поле первичного индекса будет инкрементировано средствами сервера InterBase (соответствующими генератором и триггером). Однако получить это значение в приложении можно только сохранив изменения и обновив набор данных, что зачастую требует больших затрат ресурсов.

Для решения этой проблемы в компоненте `TIBQuery` предназначено свойство

```
property GeneratorField: TIBGeneratorField;
```

Редактор свойства (рис. 22.2) позволяет связать генератор с инкрементируемым полем.

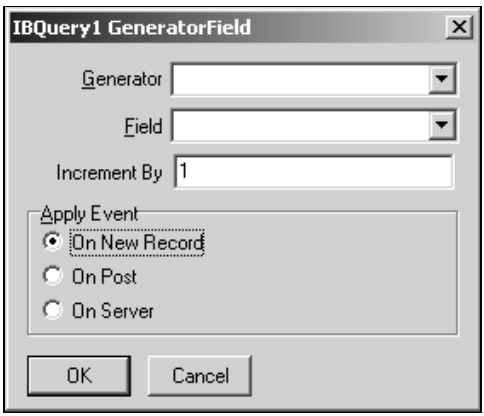


Рис. 22.2. Редактор свойства `GeneratorField` компонента `TIBQuery`

Список **Generator** позволяет выбрать один из доступных генераторов базы данных. Список **Field** задает инкрементируемое поле набора данных. В строке **Increment By** определяется шаг прибавляемого значения поля.

Группа переключателей **Apply Events** определяет событие, при котором срабатывает генератор:

- **On New Record** — при создании новой записи;
- **On Post** — при сохранении новой записи;
- **On Server** — генератор управляется сервером.

Редактор свойства `GeneratorField` попросту присваивает значения полям экземпляра класса `TIBGeneratorField`.

Методы-обработчики событий полностью соответствуют классу `TIBCustomDataSet` (см. табл. 22.2).

Компонент *TIBStoredProc*

Компонент `TIBStoredProc` полностью соответствует стандартному прототипу, описанному в *главе 16*.

Имя хранимой процедуры задается свойством

```
property StoredProcName: String;
```

А список всех доступных на этапе выполнения хранимых процедур возвращает свойство

```
property StoredProcedureNames: TStrings;
```

Параметры хранимой процедуры содержатся в стандартном свойстве

```
property Params: TParams;
```

а общее число параметров возвращает свойство

```
property ParamCount: Word;
```

Свойство

```
property Prepared: Boolean;
```

позволяет определить, подготовлена ли хранимая процедура к выполнению.

Методы-обработчики событий полностью соответствуют классу `TIBCustomDataSet` (см. табл. 22.2).

Компонент *TIBDataSet*

Компонент `TIBDataSet` предназначен для представления в приложениях наборов данных от сложных запросов. При этом набор данных остается редактируемым. Это достигается заданием дополнительных запросов на удаление, изменение и добавление данных. Однако в компоненте `TIBDataSet` интегрированы одновременно и сам сложный запрос, и вспомогательные запросы.

Основной запрос содержится в свойстве

```
property SelectSQL: TStrings;
```

Создание запроса облегчает простой редактор, вызываемый при щелчке на кнопке в поле редактирования свойства в Инспекторе объектов (рис. 22.3).

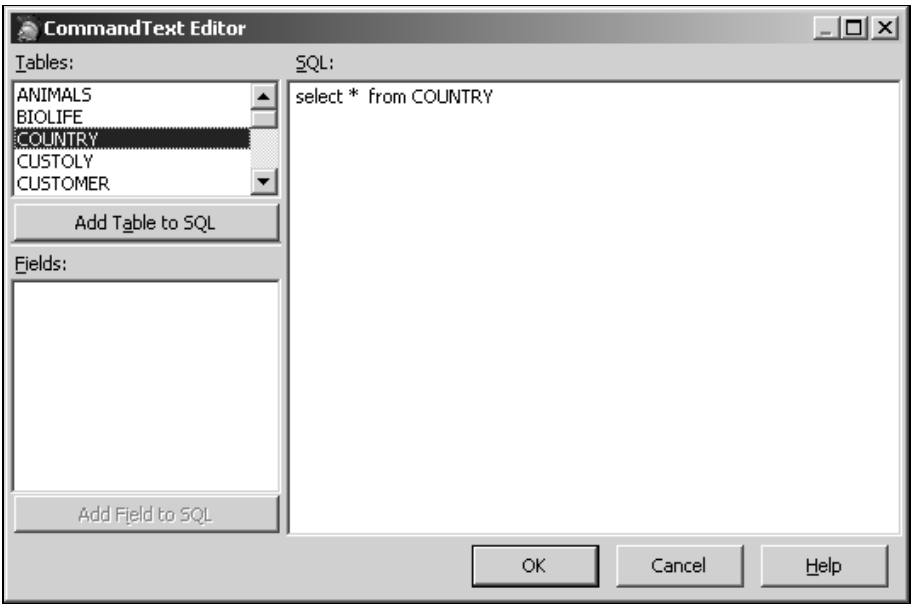


Рис. 22.3. Редактор запроса компонента `TIBDataSet`

Каждому запросу (основному и вспомогательным) соответствует собственный объект `TIBSQL`, который подробно рассматривается далее в этой главе. Свойства и методы компонента `TIBDataSet` приведены в табл. 22.5.

Таблица 22.5. *Свойства и методы компонента* `TIBDataSet`

Объявление	Тип	Описание
Свойства		
`property BufferChunks: Integer;`	Pb	Определяет число записей в буфере набора данных
`property DeleteSQL: TStrings;`	Pb	Содержит текст запроса, обеспечивающего удаление записей из набора данных
`property InsertSQL: TStrings;`	Pb	Содержит текст запроса, обеспечивающего добавление записей в набор данных

Таблица 22.5 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`property ModifySQL: TStrings;`	Pb	Содержит текст запроса, обеспечивающего изменение записей из набора данных
`property Params: TIBXSQLDA;`	Ro	Структура API, содержащая параметры запроса
`property Prepared: Boolean;`	Ro	Позволяет определить, подготовлен ли запрос к выполнению
`property QDelete: TIBSQL;`	Ro	Объект запроса на удаление
`property QInsert: TIBSQL;`	Ro	Объект запроса на добавление
`property QModify: TIBSQL;`	Ro	Объект запроса на изменение
`property QRefresh: TIBSQL;`	Ro	Объект запроса на обновление
`property QSelect: TIBSQL;`	Ro	Объект запроса на отбор данных
`property RefreshSQL: TStrings;`	Pb	Содержит текст запроса, обеспечивающего обновление записей набора данных
`property SelectSQL: TStrings;`	Pb	Содержит текст основного запроса набора данных
`type TIBSQLTypes = set of (SQLUnknown, SQLSelect, SQLInsert, SQLUpdate, SQLDelete, SQLDDL, SQLGetSegment, SQLPutSegment, SQLExecProcedure, SQLStartTransaction, SQLCommit, SQLRollback, SQLSelectForUpdate, SQLSetGenerator);`	Ro	Возвращает тип основного запроса набора данных: `SQLUnknown` — неизвестный тип; `SQLSelect`, `SQLInsert`, `SQLUpdate`, `SQLDelete` — стандартные типы; `SQLDDL` — выражение DDL; `SQLGetSegment`, `SQLPutSegment` — запросы с полями BLOB; `SQLExecProcedure`, `SQLStartTransaction`, `SQLCommit`, `SQLRollback` — обработка транзакций; `SQLSelectForUpdate` — хранимая процедура, возвращающая набор данных; `SQLSetGenerator` — выполнение генератора

Таблица 22.5 *(окончание)*

Объявление	Тип	Описание
Методы		
`procedure Prepare;`	Pu	Осуществляет подготовку всех запросов компонента к выполнению
`procedure UnPrepare;`	Pu	Возвращает все запросы набора данных к исходному состоянию
Методы-обработчики событий		
`property DatabaseDisconnected: TNotifyEvent;`	Pb	Вызывается после отключения базы данных
`property DatabaseDisconnecting: TNotifyEvent;`	Pb	Вызывается во время отключения базы данных
`property DatabaseFree: TNotifyEvent;`	Pb	Вызывается после того, как компонент соединения освобождает занимаемую память

Компонент *TIBSQL*

Компонент `TIBSQL` предназначен для быстрого выполнения запросов SQL, поэтому не обеспечивает связи с компонентами представления данных.

Для обеспечения скорости выполнения запроса из компонента удалены все дополнительные механизмы, обслуживающие набор данных. Фактически компонент `TIBSQL` не имеет отношения к обычным компонентам доступа к данным, его непосредственным предком является класс `TComponent`, а не `TDataSet`. Поэтому он только передает через компонент соединения `TIBDatabase` запрос серверу и получает назад результат выполнения запроса.

Для повышения скорости компонент не обеспечивает полноценной навигации по набору данных. Перемещение по набору данных возможно только в прямом направлении (однонаправленный курсор).

Возвращаемые набором данных текущие значения полей содержатся не в привычном наборе объектов полей `TField`, а в объекте `TIBXSQLDA`. Так как структура `XSQLDA` создается сервером при выполнении запроса, существенно уменьшается время открытия набора данных компонента. Свойства и методы компонента `TIBSQL` приведены в табл. 22.6.

Таблица 22.6. *Свойства и методы компонента* `TIBSQL`

Объявление	Тип	Описание
Свойства		
`property Bof: Boolean;`	Pu	Значение `True` говорит о том, что курсор находится в начале набора данных
`property Database: TIBDatabase;`	Pb	Определяет компонент соединения с базой данных
`property DBHandle: PISC_DB_HANDLE;`	Pu	Указатель API на объект базы данных
`property Eof: Boolean;`	Pu	Значение `True` говорит о том, что курсор находится в конце набора данных
`property FieldIndex: [FieldName: String]: Integer;`	Pu	Список порядковых номеров полей по их именам
`property Fields[const Idx: Integer]: TIBXSQLVAR;`	Pu	Индексированный список структур `XSQLVAR`, хранящих значения полей набора данных
`property GenerateParamNames: Boolean;`	Pu	Установка свойства в значение `True` проводит к созданию список имен параметров запроса в свойстве `Params`
`property GoToFirstRecordOnExecute: Boolean;`	Pb	Значение `True` обеспечивает установку курсора на первую запись набора данных при его открытии
`property Handle: TISC_STMT_HANDLE;`	Pu	Содержит указатель API на запрос
`property Open: Boolean;`	Pu	Позволяет определить, открыт ли набор данных
`property ParamCheck: Boolean;`	Pb	Позволяет определить, был ли заново сгенерирован список параметров запроса при изменении его текста во время выполнения
`property Params: TIBXSQLDA;`	Pu	Область дескрипторов запроса
`property Plan: String;`	Pu	Содержит план запроса после его подготовки
`property Prepared: Boolean;`	Pu	Значение `True` сообщает о том, что запрос готов к выполнению
`property RecordCount: Integer;`	Pu	Возвращает число записей набора данных
`property RowsAffected: Integer;`	Pu	Возвращает число записей, обработанных запросом

Таблица 22.6 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`property SQL: TStrings;`	Pb	Содержит текст запроса
`property SQLType: TIBSQLTypes read FSQLType;`	Pu	Возвращает тип запроса (подробнее см. табл. 22.5)
`property Transaction: TIBTransaction;`	Pb	Указывает на компонент транзакции
`property TRHandle: PISC_TR_HANDLE;`	Pu	Содержит указатель API на транзакцию, в которой работает запрос
`property UniqueRelationName: String;`	Pu	Возвращает уникальное внутреннее имя запроса
Методы		
`procedure BatchInput(InputObject: TIBBatchInput);`	Pu	Выполняет запрос с параметрами для переноса в объект `InputObject`
`procedure BatchOutput(OutputObject: TIBBatchOutput);`	Pu	Выполняет запрос с параметрами для переноса в объект `OutputObject`
`function Call(ErrCode: ISC_STATUS; RaiseError: Boolean): ISC_STATUS;`	Pu	Возвращает текст сообщения об ошибке по ее коду `ErrCode`
`procedure CheckClosed;`	Pu	Вызывает исключение, если набор данных открыт
`procedure CheckOpen;`	Pu	Вызывает исключение, если набор данных закрыт
`procedure CheckValidStatement;`	Pu	Вызывает исключение, если запрос некорректен
`procedure Close;`	Pu	Закрывает набор данных
`function Current: TIBXSQLDA;`	Pu	Ссылка на область дескрипторов запроса
`procedure ExecQuery;`	Pu	Выполняет запрос
`function FieldByName[FieldName: String]: TIBXSQLVAR;`	Pu	Возвращает структуру `XSQLVAR` по имени поля
`procedure FreeHandle;`	Pu	Освобождает ресурсы, занятые запросом

Таблица 22.6 *(окончание)*

Объявление	Тип	Описание
Методы		
`function Next: TIBXSQLDA;`	Pu	Возвращает область дескрипторов для следующей записи
`procedure Prepare;`	Pu	Готовит запрос к выполнению
Методы-обработчики событий		
`property OnSQLChanging: TNotifyEvent;`	Pb	Вызывается при изменении запроса

Текст запроса задается обычным для всех компонентов запросов свойством `SQL`. Для выполнения запроса служит также знакомое свойство `ExecSQL`. После этого можно обращаться к созданному компонентом набору данных.

Значения полей из текущей записи доступны через свойство `Fields`. Обратите внимание, что это не объекты типа `TFields`, а структуры `XSQLVAR` из области дескрипторов.

Будут ли переданы значения полей в компонент, зависит от значения свойства `GoToFirstRecordOnExecute`.

Доступ к области дескрипторов осуществляется через свойство `Current`.

Переход к следующей записи осуществляется методом `Next`. При этом обновляется область дескрипторов запроса.

Обработка событий

Клиентское приложение Delphi, работающее с сервером InterBase, имеет возможность отслеживать события, происходящие в базе данных и вызываемые другими процессами или приложениями. Для этого используется компонент `TIBEvents`. Он позволяет определить список необходимых событий и предоставляет разработчику простой механизм отслеживания возникающих на сервере событий.

Список событий задается свойством

```
property Events: TStrings;
```

в котором можно определить до 15 контролируемых событий.

Выбранные события необходимо зарегистрировать на сервере. Для этого применяется метод

```
procedure RegisterEvents;
```

Метод

```
procedure QueueEvents;
```

начинает процесс передачи сообщений от сервера.

При возникновении на сервере зарегистрированного события компонент вызывает метод-обработчик события

```
property OnEventAlert: TEventAlert;
TEventAlert = procedure( Sender: TObject; EventName: String;
EventCount: longint; var CancelAlerts: Boolean)
```

Назначение параметров здесь следующее:

- `EventName` — содержит имя последнего произошедшего события;
- `EventCount` — содержит число заданных событий, произошедших с момента последнего вызова метода-обработчика;
- `CancelAlerts` — позволяет прервать процесс передачи сообщений приложению. Для этого необходимо присвоить параметру значение `True`.

Для возобновления работы компонента нужно снова вызвать метод `QueueEvents`. Свойства и методы компонента `TIBEvents` приведены в табл. 22.7.

Таблица 22.7. *Свойства и методы компонента* `TIBEvents`

Объявление	Тип	Описание
Свойства		
`property Database: TIBDatabase;`	Pb	Задает базу данных
`property Events: TStrings;`	Pb	Список контролируемых событий
`property Queued: Boolean;`	Ro	Значение `True` говорит о том, что процесс передачи сообщений работает
`property Registered: Boolean;`	Pb	Определяет регистрацию сообщений на сервере
Методы		
`procedure CancelEvents;`	Pu	Останавливает процесс передачи сообщений
`procedure QueueEvents;`	Pu	Включает процесс передачи сообщений
`procedure RegisterEvents;`	Pu	Проводит регистрацию сообщений на сервере
`procedure UnRegisterEvents;`	Pu	Отменяет регистрацию сообщений на сервере

Таблица 22.7 *(окончание)*

Объявление	Тип	Описание
Методы-обработчики событий		

`property OnEventAlert: TEventAlert;`
 	
`TEventAlert = procedure( Sender: TObject; EventName: String; EventCount: longint; var CancelAlerts: Boolean)`
	Pb	Вызывается при передаче сообщения от сервера компоненту

Информация о состоянии базы данных

В процессе отладки и выполнения клиентских приложений для сервера InterBase разработчик может получать подробную информацию об этих процессах.

Компонент `TIBDatabaseInfo` предоставляет информацию о текущем состоянии базы данных.

Компонент `TIBSQLMonitor` отслеживает выполнение запросов на сервере.

Компонент *TIBDatabaseInfo*

Компонент `TIBDatabaseInfo` обладает большим числом свойств, содержащих разнообразные сведения о состоянии БД. Компонент очень прост в применении.

Выбор базы данных (компонента `TIBDatabase`) осуществляется стандартным свойством

```
property Database: TIBDatabase;
```

В процессе работы с базой данных свойствам компонента `TIBDatabaseInfo` передаются соответствующие значения. Разработчику необходимо лишь в нужных местах установить значения нужных свойств. Свойства и методы компонента `TIBDatabaseInfo` приведены в табл. 22.8.

Таблица 22.8. *Свойства и методы компонента* `TIBDatabaseInfo`

Объявление	Тип	Описание
Свойства		
`property Allocation: Long;`	Ro	Число выделенных страниц БД

Таблица 22.8 *(продолжение)*

Объявление	Тип	Описание
Свойства		
`property BackoutCount: TStringList;`	Ro	Число вариантов удаленных записей
`property BaseLevel: Long;`	Ro	Версия базы данных (содержится во втором байте)
`property CurrentMemory: Long;`	Ro	Объем памяти (в байтах), занятый сервером
`property Database: TIBDatabase;`	Pb	Ссылка на компонент соединения с БД
`property DBFileName: String;`	Ro	Имя файла БД
`property DBImplementationClass: Long;`	Ro	Номер класса описания
`property DBImplementationNo: Long;`	Ro	Номер описания
`property DBSiteName: String;`	Ro	Имя сайта БД
`property DBSQLDialect: Long;`	Ro	Номер диалекта SQL
`property DeleteCount: TStringList;`	Ro	Число удалений с момента последнего обновления БД
`property ExpungeCount: TStringList;`	Ro	Число удалений записей с последнего сохранения БД
`property Fetches: Long;`	Ro	Число чтений из кэша
`property ForcedWrites: Long;`	Ro	Режим чтения: 0 — асинхронное чтение; 1 — синхронное чтение
`property InsertCount: TStringList;`	Ro	Число добавлений в БД с последнего сохранения
`property Marks: Long;`	Ro	Число выполненных записей в кэш
`property MaxMemory: Long;`	Ro	Максимальный размер памяти, занимаемый БД с момента последнего сохранения
`property NoReserve: Long;`	Ro	Резервирование страниц: 0 — резервирование есть; 1 — резервирования нет
`property NumBuffers: Long;`	Ro	Число выделенных буферов

Таблица 22.8 *(окончание)*

Объявление	Тип	Описание
Свойства		
`property ODSMajorVersion: Long;`	Ro	Верхнее значение ODS
`property ODSMinorVersion: Long;`	Ro	Нижнее значение ODS
`property PageSize: Long;`	Ro	Размер страницы БД
`property PurgeCount: TStringList;`	Ro	Общее число удаленных по любой причине записей
`property ReadIdxCount: TStringList;`	Ro	Число чтений через индексы с последнего сохранения
`property ReadOnly: Long;`	Ro	0 — БД только для чтения; 1 — перезаписываемая БД
`property Reads: Long;`	Ro	Число чтений из БД
`property ReadSeqCount: TStringList;`	Ro	Число чтений таблиц целиком с последнего сохранения
`property SweepInterval: Long;`	Ro	Число зафиксированных транзакций
`property UpdateCount: TStringList;`	Ro	Число обновлений БД с последнего сохранения
`property UserNames: TStringList;`	Ro	Список активных пользователей
`property Version: String;`	Ro	Версия БД
`property Writes: Long;`	Ro	Число постраничных записей
Методы		
`function Call(ErrCode: ISC_STATUS; RaiseError: Boolean): ISC_STATUS;`	Pu	Возвращает сообщение об ошибке по параметру `ErrCode`

Компонент *TIBSQLMonitor*

Компонент `TIBSQLMonitor` позволяет получать в клиентском приложении сообщения от сервера о выполняемых им операциях. Для этого используется метод-обработчик компонента

```
TSQLEvent = procedure(EventText: String) of object;
property OnSQL: TSQLEvent;
```

Параметр `EventText` содержит текст сообщения.

В компоненте соединения с БД можно установить перечень событий сервера, на которые будет реагировать компонент `TIBSQLMonitor`. Это делается при помощи свойства `TraceFlags`. Возможные значения множества означают контроль за следующими операциями:

- `tfQPrepare` — подготовка запроса к выполнению (вызов метода `Prepare`);
- `tfQExecute` — выполнение запроса (вызов метода `ExecSQL`);
- `tfQFetch` — вызов запроса (вызов методов `Open`, `Close`);
- `tfError` — возникновение ошибки;
- `tfStmt` — все операции с запросами;
- `tfConnect` — подключение и отключение БД;
- `tfTransact` — выполнение транзакций;
- `tfBlob` — операции с данными BLOB;
- `tfService` — вспомогательные операции;
- `tfMisc` — любые операции, не учтенные вышеперечисленными значениями.

Резюме

В этой главе рассмотрены возможности набора компонентов InterBase Express. Они обеспечивают быстрый и эффективный доступ к базам данных на серверах InterBase. Для доступа к данным этим компонентам не требуется BDE, они используют только возможности API InterBase.

Часть компонентов обеспечивает быстрый переход со стандартных компонентов, инкапсулирующих набор данных, и повторяют функциональность компонентов `TTable`, `TQuery`, `TStoredProc` и т. д.

Компоненты `TIBSQL` и `TIBDataSet` полностью основаны на механизмах API InterBase, функционируют еще эффективнее, но требуют нестандартных приемов работы.

Глава 23

Технология dbGo

Технология Microsoft ActiveX Data Objects (ADO) завоевала популярность у разработчиков благодаря универсальности: базовый набор интерфейсов OLE DB имеется в каждой современной операционной системе Microsoft. Поэтому для обеспечения доступа приложения к данным достаточно лишь правильно указать провайдер соединения ADO и затем переносить программу на любой компьютер, где имеется требуемая база данных и, конечно, установленная ADO.

В Delphi технология доступа к данным на основе ADO носит название dbGo.

В Палитре инструментов Delphi имеется категория **dbGo**, содержащая набор компонентов, позволяющих создавать полноценные приложения БД, обращающиеся к данным через ADO.

В этой главе рассматриваются следующие вопросы.

- Создание соединения с базой данных через ADO в приложении Delphi.
- Применение объекта набора записей ADO в приложении.
- Использование таблиц, запросов SQL и хранимых процедур.
- Команды и объекты команды ADO.

Основы ADO

Технология Microsoft ActiveX Data Objects обеспечивает универсальный доступ к источникам данных из приложений БД. Такую возможность предоставляют функции набора интерфейсов, созданного на основе общей модели объектов COM, и описанного в спецификации OLE DB.

Технология ADO и интерфейсы OLE DB обеспечивают для приложений единый способ доступа к источникам данных различных типов. Например, приложение, использующее ADO, может применять одинаково сложные операции и к данным, хранящимся на корпоративном сервере SQL, и к

электронным таблицам, и к локальным СУБД. Запрос SQL, направленный любому источнику данных через ADO, будет выполнен.

Возникает вопрос: каким образом источники данных смогут выполнить этот запрос?

За серверы БД беспокоится не стоит, обработка запросов SQL — это их основная обязанность. Но как быть с файловыми последовательностями, электронными таблицами, файлами электронной почты и т. д.? Здесь на помощь приходят механизмы ADO и интерфейсы OLE DB.

OLE DB представляет собой набор специализированных объектов COM, инкапсулирующих стандартные функции обработки данных и специализированные функции конкретных источников данных, и интерфейсов, обеспечивающих передачу данных между объектами (рис. 23.1).

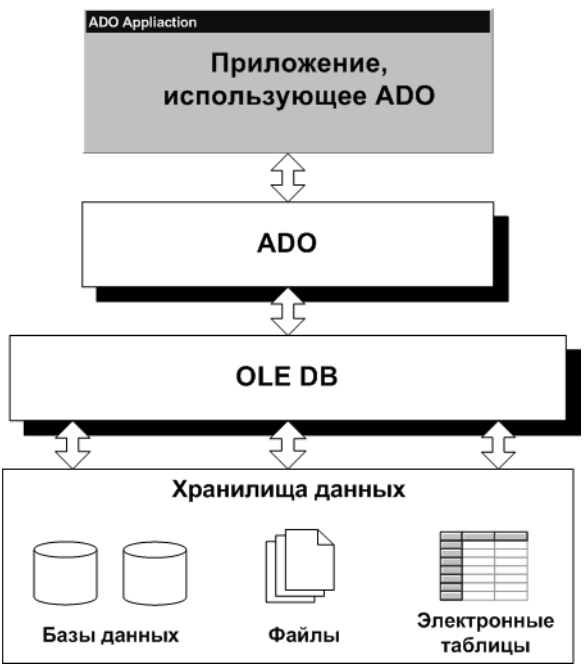


Рис. 23.1. Схема доступа к данным через ADO

Согласно терминологии ADO, любой источник данных (база данных, электронная таблица, файл) называется хранилищем данных, с которым при посредстве провайдера данных взаимодействует приложение. Минимальный набор компонентов приложения может включать объект соединения, объект набора данных, объект процессора запросов.

Примечание

Объекты OLE DB создаются и функционируют так же, как и другие объекты COM. Каждому объекту соответствует идентификатор класса `CLSID`, хранящийся в системном реестре. Для создания объекта используются метод `CoCreateInstance` и соответствующая фабрика класса. Объекту соответствует набор интерфейсов, к методам которых можно обращаться после создания объекта.

В результате приложение обращается не непосредственно к источнику данных, а к объекту OLE DB, который "умеет" представить данные (например, из файла электронной почты) в виде таблицы БД или результата выполнения запроса SQL.

Технология ADO в целом включает в себя не только сами объекты OLE DB, но и механизмы, обеспечивающие взаимодействие объектов с данными и приложениями. На этом уровне важнейшую роль играют провайдеры ADO, координирующие работу приложений с хранилищами данных различных типов.

Такая архитектура позволяет сделать набор объектов и интерфейсов открытым и расширяемым. Набор объектов и соответствующий провайдер может быть создан для любого хранилища данных без внесения изменений в исходную структуру ADO. При этом существенно расширяется само понятие данных — ведь можно разработать набор объектов и интерфейсов и для нетрадиционных табличных данных. Например, это могут быть графические данные геоинформационных систем, древовидные структуры из системных реестров, данные CASE-инструментов и т. д.

Так как технология ADO основана на стандартных интерфейсах COM, которые являются системным механизмом Windows, это сокращает общий объем работающего программного кода и позволяет распространять приложения БД без вспомогательных программ и библиотек.

Примечание

Нижеследующее описание спецификации OLE DB представлено в соответствии с официальной терминологией Microsoft для данной предметной области.

Спецификация OLE DB различает следующие типы объектов, которые будут рассмотрены далее.

- Перечислитель (Enumerator) — выполняет поиск источников данных или других перечислителей. Используется для обеспечения функционирования провайдеров ADO.
- Объект — источник данных (Data Source Object) — представляет хранилище данных.

- Сессия (Session) — объединяет совокупность объектов, обращающихся к одному хранилищу данных.
- Транзакция (Trasaction) — инкапсулирует механизм выполнения транзакции.
- Команда (Command) — содержит текст команды и обеспечивает ее выполнение. Командой может быть запрос SQL, обращение к таблице БД и т. д.
- Набор рядов (Rowset) — представляет собой совокупность строк данных, являющихся результатом выполнения команды ADO.
- Объект-ошибка (Error) — содержит информацию об исключительной ситуации.

Рассмотрим функциональные возможности основных объектов и интерфейсов OLE DB.

Перечислители

Объекты-перечислители обеспечивают поиск любых объектов ADO, которые осуществляют доступ к источникам данных. При этом другие перечислители также видны в данном перечислителе.

Первичный поиск источников данных осуществляется в провайдере ADO. Перечислители могут отбирать только источники данных конкретных типов, поэтому провайдер обеспечивает доступ к конкретному типу хранилища данных.

В составе ADO имеется системный корневой перечислитель, который выполняет начальный поиск других перечислителей и источников данных. Его можно использовать, зная его идентификатор класса `CLSID_OLEDB_ENUMERATOR`.

Примечание

В Delphi GUID глобального перечислителя содержится в файле ..\Source\Win32\db\OleDB.pas.

```
CLSID_OLEDB_ENUMERATOR: TGUID = '{C8B522D0-5CF3-11CE-ADE5-
00AA0044773D}';
```

Функции перечислителя содержатся в интерфейсе `ISourcesRowset`.

Метод

```
function GetSourcesRowset(const punkOuter: IUnknown; const riid:
TGUID; cPropertySets: UINT; rgProperties: PDBPropSetArray; out
ppSourcesRowset: IUnknown): HResult; stdcall;
```

возвращает ссылку на объект набора рядов, содержащий сведения о найденных источниках данных или перечислителях.

Объекты соединения с источниками данных

Внутренний механизм ADO, обеспечивающий соединение с хранилищем данных, использует два типа объектов. Это объекты — источники данных и объекты-сессии.

Объект — источник данных обеспечивает представление информации о требуемом реальном источнике данных и подключение к нему.

Для ввода сведений о хранилище данных используется интерфейс `IDBProperties`. Для успешного подключения необходимо задать обязательные сведения. Вероятно, для любого хранилища данных будет актуальной информация об его имени, пользователе и пароле. Однако каждый тип хранилища имеет собственные уникальные настройки. Для получения списка всех обязательных параметров соединения с данным хранилищем можно воспользоваться методом `function GetPropertyInfo(cPropertyIDSets: UINT; rgPropertyIDSets: PDBPropIDSetArray; var pcPropertyInfoSets: UINT; out prgPropertyInfoSets: PDBPropInfoSet; ppDescBuffer: PPOleStr): HResult; stdcall;`

который возвращает заполненную структуру `DBPROPINFO`.

```
PDBPropInfo = ^TDBPropInfo;
  DBPROPINFO = packed record
    pwszDescription: PWideChar;
    dwPropertyID: DBPROPID;
    dwFlags: DBPROPFLAGS;
    vtType: Word;
    vValues: OleVariant;
  end;
  TDBPropInfo = DBPROPINFO;
```

Для каждого обязательного параметра в элементе `dwFlags` устанавливается значение `DBPROPFLAGS_REQUIRED`.

Для инициализации соединения необходимо использовать метод

```
function Initialize: HResult; stdcall;
```

интерфейса `IDBInitialize` объекта — источника данных.

Сессия

Из объекта — источника данных можно создавать объекты-сессии. Для этого предусмотрен метод

```
function CreateSession(const punkOuter: IUnknown; const riid: TGUID;
out ppDBSession: IUnknown): HResult; stdcall;
```

интерфейса `IDBCreateSession`. Сессия предназначена для обеспечения работы транзакций и наборов рядов.

Транзакции

Управление транзакциями в OLE DB реализовано на двух уровнях.

Во-первых, всеми необходимыми методами обладает объект сессии. Он имеет интерфейсы `ITransaction`, `ITransactionJoin`, `ITransactionLocal` и `ITransactionObject`.

Внутри сессии транзакция управляется интерфейсами `ITransactionLocal`, `ItransactionSC`, `ITransaction` и их методами `StartTransaction`, `Commit`, `Rollback`.

Во-вторых, для объекта сессии можно создать объект транзакции при помощи метода `function GetTransactionObject(ulTransactionLevel: UINT; out ppTransactionObject: ITransaction): HResult; stdcall;`

интерфейса `ITransactionObject`, который возвращает ссылку на интерфейс объекта-транзакции.

Наборы рядов

Объект "набор рядов" является основным объектом ADO, обеспечивающим работу с данными. Он инкапсулирует совокупность рядов из источника данных и механизмы навигации по рядам и поддержания рядов в актуальном состоянии.

Объект сессии имеет обязательный интерфейс `IOpenRowset` с методом

```
function OpenRowset(const punkOuter: IUnknown; pTableID: PDBID;
pIndexID: PDBID; const riid: TGUID; cPropertySets: UINT;
rgPropertySets: PDBPropSetArray; ppRowset: PIUnknown): HResult;
stdcall;
```

который открывает необходимый набор рядов.

В зависимости от возможностей источника данных набор рядов может поддерживать различные интерфейсы. Но пять из них являются обязательными:

- `IRowset` — обеспечивает навигацию по рядам;
- `IAccessor` — обеспечивает представление информации о формате рядов, содержащихся в буфере набора рядов;
- `IRowsetInfo` — позволяет получить информацию о наборах рядов (например, число рядов или число обновленных рядов);
- `IColumnsInfo` — позволяет получить информацию о колонках рядов (наименование, тип данных, возможность обновления и т. д.);

- `IConvertType` — содержит единственный метод `CanConvert`, позволяющий определить возможность преобразования типов данных в наборе рядов.

> **Примечание**
>
> В отличие от привычной практики разработки интерфейсов в рамках модели COM, интерфейсы OLE DB часто имеют всего один-два метода. В результате большая группа интерфейсов реализует несколько вполне стандартных функций.

Дополнительные возможности по управлению набором рядов предоставляют следующие интерфейсы.

- `IRowsetChange` — выполняет изменения в наборе рядов (вносит изменения, добавляет новые ряды, удаляет ряды и т. д.).
- `IRowsetIdentity` — позволяет сравнивать ряды разных рядов.
- `IRowsetIndex` — обеспечивает использование индексов.
- `IRowsetLocate` — выполняет поиск в наборе рядов.
- `IRowsetUpdate` — реализует механизм кэширования изменений.

Команды

Программные средства ADO были бы неполными, если бы не имели возможности использовать для работы с данными язык SQL. Операторы DML и DDL, ряд специальных операторов ADO носят общее название текстовых команд.

Объект-команда инкапсулирует саму текстовую команду и механизм обработки и передачи команды. Объект-команда выполняет следующие операции:

- разбор текста команды;
- связывание команды с источником данных;
- оптимизация команды;
- передача команды источнику данных.

Главный интерфейс объекта команды `ICommand` имеет три метода:

- `function Cancel: HResult; stdcall;`

  отменяет выполнение команды;

- `function Execute(const punkOuter: IUnknown; const riid: TGUID; var pParams: DBPARAMS;`

  `pcRowsAffected: PInteger; ppRowset: PIUnknown): HResult; stdcall;`

  исполняет команду;

- `function GetDBSession(const riid: TGUID; out ppSession: IUnknown): HResult; stdcall;`

  возвращает ссылку на интерфейс сессии, вызвавший данную команду.

Помимо основного, объект команды обеспечивает доступ к дополнительным интерфейсам:

`ICommandPrepare` содержит два метода (`Prepare` и `Unprepare`) для подготовки команды.

`ICommandProperties` задает для команды свойства, которые должны поддерживаться возвращаемым командой набором данных.

`ICommandText` управляет текстом команды (этот интерфейс обязателен для объекта команды).

`ICommandWithParameters` обеспечивает работу с параметрами команды.

Провайдеры ADO

Провайдеры ADO обеспечивают соединение приложения, использующего данные через ADO, с источником данных (сервером SQL, локальной СУБД, файловой системой и т. д.). Для каждого типа хранилища данных должен существовать провайдер ADO.

Провайдер "знает" о местоположении хранилища данных и его содержании, умеет обращаться к данным с запросами и интерпретировать возвращаемую служебную информацию и результаты запросов с целью их передачи приложению.

Список установленных в данной операционной системе провайдеров доступен для выбора при установке соединения через компонент `TADOConnection`.

При инсталляции Microsoft ActiveX Data Objects в операционной системе устанавливаются следующие стандартные провайдеры:

- Microsoft Jet OLE DB Provider — реализует соединение с данными СУБД Access при посредстве технологии DAO;
- Microsoft OLE DB Provider for Microsoft Indexing Service — обеспечивает доступ только для чтения к файлам и интернет-ресурсам Microsoft Indexing Service;
- Microsoft OLE DB Provider for Microsoft Active Directory Service — обеспечивает доступ к ресурсам службы каталогов (Active Directory Service);
- Microsoft OLE DB Provider for Internet Publishing — позволяет использвать ресурсы, предоставляемые Microsoft FrontPage, Microsoft Internet Information Server, HTTP-файлы;
- Microsoft Data Shaping Service for OLE DB — позволяет работать с иерархическими наборами данных;

- Microsoft OLE DB Simple Provider — предназначен для организации доступа к источникам данных, поддерживающим только базисные возможности OLE DB;
- Microsoft OLE DB Provider for ODBC drivers — обеспечивает доступ к данным, которые уже "прописаны" при помощи драйверов ODBC. Однако реальное использование столь экзотичных вариантов соединений представляется проблематичным. Драйверы ODBC и так славятся своей медлительностью, поэтому дополнительный слой сервисов здесь ни к чему;
- Microsoft OLE DB Provider for Oracle — реализует соединение с сервером Oracle;
- Microsoft OLE DB Provider for SQL Server — реализует соединение с сервером Microsoft SQL Server.

Технология dbGo

Механизм доступа к данным через ADO и многочисленные объекты и интерфейсы реализованы в VCL Delphi в виде набора компонентов dbGo, расположенных в одноименной категории Палитры инструментов. Все необходимые интерфейсы, обеспечивающие работу компонентов, объявлены и описаны в файлах OleDB.pas и ADODB.pas в папке ..\Source\Win32\db.

Компоненты ADO

Компонент `TADOConnection` вобрал возможности перечислителя, источника данных и сессии с возможностями обслуживания транзакций.

Текстовые команды ADO реализованы в компоненте `TADOCommand`.

Наборы рядов (нотация Microsoft) можно получить при помощи компонентов `TADOTable`, `TADOQuery` и `TADOStoredProc`. Каждый из них реализует способ доступа к конкретному типу представления данных в хранилище. Далее по тексту применительно к компонентам Delphi совокупность возвращаемых из хранилища данных строк будем называть набором записей, что соответствует документации Borland (см. **www.borland.com** или **www.borland.ru**) и стилю изложения предыдущих глав.

Набор свойств и методов компонентов ADO обеспечивает реализацию всех необходимых приложению БД функций. Способы использования компонентов dbGo немногим отличаются от стандартных компонентов VCL доступа к данным *(см. главу 16)*.

Однако при необходимости разработчик может задействовать все возможности интерфейсов ADO, обращаясь к ним через соответствующие объекты ADO. Ссылки на объекты имеются в компонентах.

Механизм соединения с хранилищем данных ADO

Существуют два варианта подключения компонентов доступа к хранилищу данных ADO: стандартный метод ADO и стандартный метод Delphi.

В первом случае компоненты используют свойство `ConnectionString` для прямого обращения к хранилищу данных. Во втором случае применяется специальный компонент `TADOConnection`, который обеспечивает расширенное управление соединением и позволяет обращаться к данным нескольким компонентам одновременно.

Свойство `ConnectionString` предназначено для хранения информации о соединении с объектом ADO. В нем через точку с запятой перечисляются все необходимые параметры. Как минимум это должны быть имена провайдера соединения или удаленного сервера:

```
ConnectionString:='Remote Server=ServerName;Provider=ProviderName';
```

При необходимости указываются путь к удаленному провайдеру:

```
ConnectionString:='Remote Provider=ProviderName';
```

и параметры, необходимые провайдеру:

```
'User Name=User_Name;Password=Password'
```

Каждый компонент, обращающийся к хранилищу данных ADO самостоятельно, задавая параметры соединения в свойстве `ConnectionString`, открывает собственное соединение. Чем больше приложение содержит компонентов ADO, тем больше соединений может быть открыто одновременно.

Поэтому целесообразно реализовать механизм соединения ADO через специальный компонент — `TADOConnection`, который открывает соединение, заданное свойством `ConnectionString`, и предоставляет разработчику дополнительные средства управления соединением.

Компоненты, работающие с хранилищем данных ADO через данное соединение, подключаются к компоненту `TADOConnection` при помощи свойства

```
property Connection: TADOConnection;
```

которое имеет каждый компонент, инкапсулирующий набор данных ADO.

Компонент *TADOConnection*

Компонент `TADOConnection` предназначен для управления соединением с объектами хранилища данных ADO. Он обеспечивает доступ к хранилищу данных компонентам ADO, инкапсулирующим набор данных.

Применение этого компонента дает разработчику ряд преимуществ:

- все компоненты доступа к данным ADO обращаются к хранилищу данных через одно соединение;
- можно напрямую задать объект провайдера соединения;
- есть доступ к объекту соединения ADO;
- можно выполнять команды ADO;
- выполнение транзакций;
- расширенное управление соединением при помощи методов-обработчиков событий.

Настройка соединения

Перед открытием соединения необходимо задать его параметры. Для этого предназначено свойство

```
property ConnectionString: WideString;
```

которое подробно рассматривается в предыдущем пункте. Добавим лишь, что набор параметров изменяется в зависимости от типа провайдера и может настраиваться как вручную, так и при помощи специального редактора параметров соединения, который вызывается двойным щелчком на компоненте `TADOConnection`, перенесенным на форму или щелчком на кнопке в поле редактирования свойства `ConnectionString` в Инспекторе объектов (рис. 23.2).

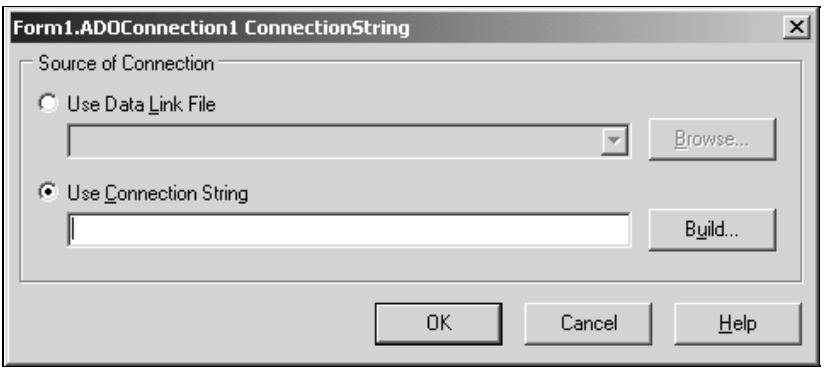


Рис. 23.2. Редактор настройки соединения ADO

Здесь можно настроить соединение через свойство `ConnectionString` (переключатель **Use Connection String**) или загрузить параметры соединения из файла с расширением udl (переключатель **Use Data Link File**).

Файл UDL (листинг 23.1) представляет собой обычный текстовый файл, в котором указываются название параметра и через знак равенства его значение. Параметры разделяются точкой с запятой.

Листинг 23.1. Демонстрационный файл DBDEMOS.UDL

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\Common
Files\Borland Shared\Data\DBDEMOS.mdb
```

Если файл параметров соединения отсутствует, настройку придется выполнять вручную. Для этого следует нажать кнопку **Build**. В результате появляется диалог **Data Link Properties**, в котором можно настроить параметры соединения (рис. 23.3). Он представляет собой четырехстраничный блокнот, позволяющий вам этап за этапом задать все необходимые параметры.

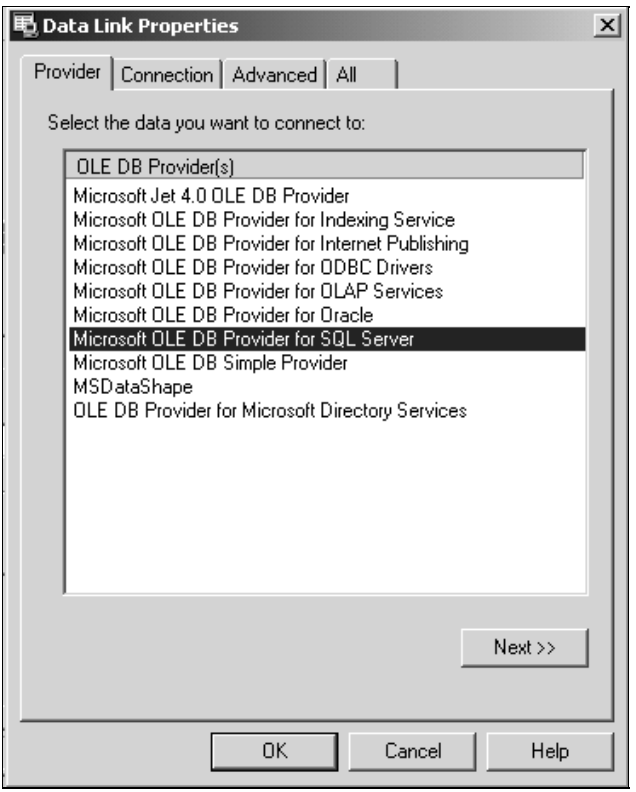


Рис. 23.3. Диалог настройки параметров соединения на странице выбора провайдера

Первая страница **Provider** позволяет выбрать провайдер OLE DB для конкретного типа источника данных из числа провайдеров, установленных в системе. Здесь вы видите провайдеры не только для серверов БД, но и служб, установленных в операционной системе. Состав элементов управления следующих страниц зависит от типа источника данных, но различается не так уж сильно. Далее практически везде необходимо задать источник данных (имя сервера, базу данных, файл и т. д.), режим аутенификации пользователя, а также определить имя и пароль пользователя.

Рассмотрим процесс настройки на примере провайдера OLE DB для сервера Microsoft SQL Server.

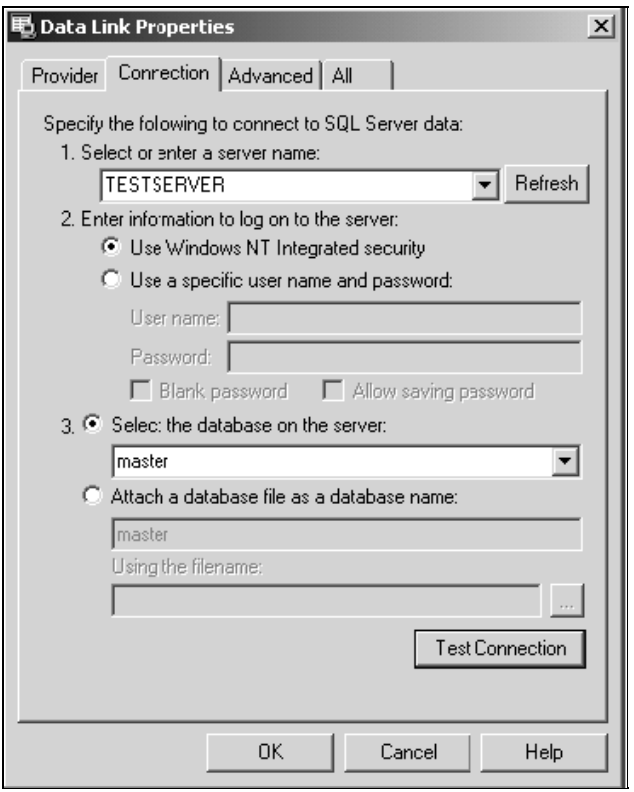


Рис. 23.4. Диалог настройки параметров соединения на странице выбора источника данных

Следующая страница **Connection** (рис. 23.4) настраивает источник данных.

На первом этапе требуется выбрать имя сервера из доступных для данного компьютера.

Второй этап определяет режим аутентификации пользователя. Это либо система безопасности Windows, либо собственная система аутентификации сервера. Здесь же надо определить имя и пароль пользователя.

Третий этап предназначен для выбора базы данных сервера.

После окончания настройки источника данных вы можете проверить соединение, нажав кнопку **Test Connection**.

Теперь перейдем на следующую страницу.

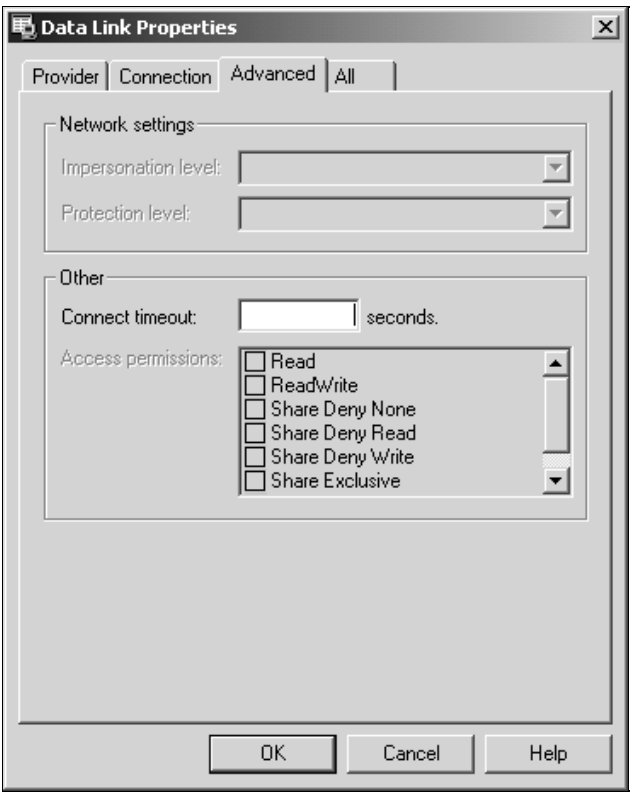


Рис. 23.5. Диалог настройки параметров соединения на странице дополнительных параметров

Страница **Advanced** (рис. 23.5) задает дополнительные параметры соединения. В зависимости от типа хранилища данных некоторые элементы этой страницы могут быть недоступны.

Список **Impersonation level** определяет возможности клиентов при подключении в соответствии с полномочиями их ролей. Здесь могут быть выбраны следующие значения.

- **Anonymous** — роль клиента недоступна серверу.
- **Identify** — роль клиента опознается сервером, но доступ к системным объектам заблокирован.
- **Impersonate** — процесс сервера может быть представлен защищенным контекстом клиента.
- **Delegate** — процесс сервера может быть представлен защищенным контекстом клиента, при этом сервер может осуществлять другие подключения.

Список **Protection Level** позволяет задать уровень защиты данных. В списке могут быть выбраны следующие значения.

- **None** — подтверждение не требуется.
- **Connect** — подтверждение необходимо только при подключении.
- **Call** — подтверждение источника данных при каждом запросе.
- **Pkt** — подтверждение получения от клиента всех данных.
- **Pkt Integrity** — подтверждение получения от клиента всех данных с соблюдением целостности.
- **Pkt Privacy** — подтверждение получения от клиента всех данных с соблюдением целостности и защита шифрованием.

В поле **Connect Timeout** можно задать время ожидания соединения в секундах. По истечении этого времени процесс прерывается.

При необходимости список **Access Permissions** задает права доступа к отдельным видам выполняемых операций. Возможны следующие значения.

- **Read** — только чтение.
- **ReadWrite** — чтение и запись.
- **Share Deny None** — полный доступ всем на чтение и запись.
- **Share Deny Read** — чтение запрещено всем.
- **Share Deny Write** — запись запрещена всем.
- **Share Exclusive** — чтение и запись запрещена всем.
- **Write** — только запись.

Последняя страница — **All** (рис. 23.6) — позволяет просмотреть и при необходимости изменить все сделанные настройки (для этого предназначена кнопка **Edit Value**) для выбранного провайдера.

После подтверждения сделанных в диалоге настроек из них формируется значение свойства `ConnectionString`.

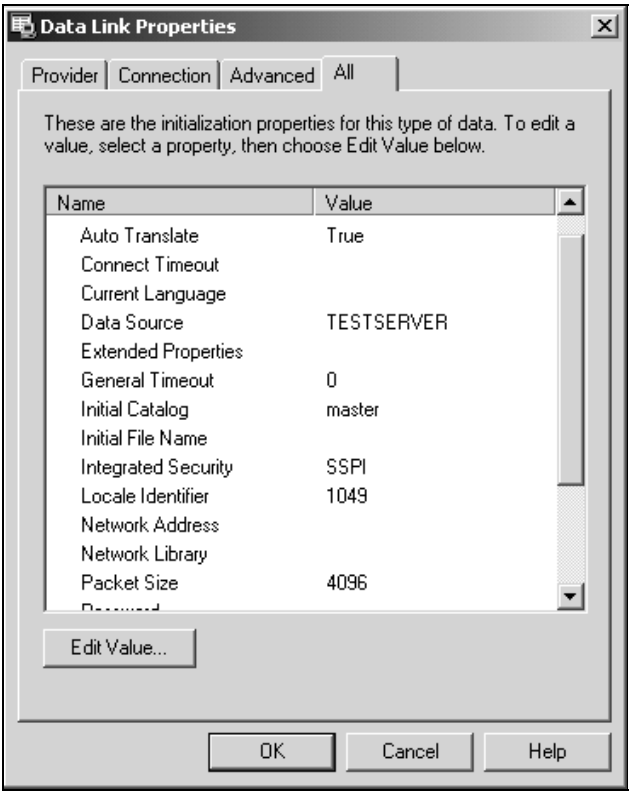


Рис. 23.6. Диалог настройки параметров соединения на странице просмотра настроек

Управление соединением

Соединение с хранилищем данных ADO открывается и закрывается при помощи свойства

```
property Connected: Boolean;
```

или методов

```
procedure Open; overload;
procedure Open(const UserID: WideString; const Password:
WideString); overload;
```

и

```
procedure Close;
```

Метод `Open` является перегружаемым при необходимости создать удаленное или локальное соединение. Для удаленного соединения применяется вариант с параметрами `UserID` и `Password`.

До и после открытия и закрытия соединения разработчику доступны соответствующие стандартные методы-обработчики событий:

- `property BeforeConnect: TNotifyEvent;`
- `property BeforeDisconnect: TNotifyEvent;`
- `property AfterConnect: TNotifyEvent;`
- `property AfterDisconnect: TNotifyEvent;`

Кроме этого, компонент `TADOConnection` имеет дополнительные методы-обработчики. После получения подтверждения от провайдера о том, что соединение будет открыто, и перед его реальным открытием вызывается метод

```
TWillConnectEvent = procedure(Connection: TADOConnection; var
ConnectionString, UserID, Password: WideString; var ConnectOptions:
TConnectOption; var EventStatus: TEventStatus) of object;
property OnWillConnect: TWillConnectEvent;
```

Параметр `Connection` содержит указатель на компонент, вызвавший обработчик.

Параметры `ConnectionString`, `UserID` и `Password` определяют строку параметров, имя и пароль пользователя.

Соединение может быть синхронным или асинхронным, что и определяется параметром `ConnectOptions` типа `TConnectOption`:

```
type TConnectOption = (coConnectUnspecified, coAsyncConnect);
```

- `coConnectUnspecified` — синхронное соединение всегда ожидает результат последнего запроса;
- `coAsyncConnect` — асинхронное соединение может выполнять новые запросы, не дожидаясь ответа от предыдущих запросов.

Наконец параметр `EventStatus` позволяет определить успешность выполнения посланного запроса на соединение:

```
type
  TEventStatus = (esOK, esErrorsOccured, esCantDeny, esCancel,
  esUnwantedEvent);
```

Здесь

- `esOK` — запрос на соединение выполнен успешно;
- `esErrorsOccured` — в процессе выполнения запроса возникла ошибка;
- `esCantDeny` — соединение не может быть прервано;

- `esCancel` — соединение было прервано до открытия;
- `esUnwantedEvent` — внутренний флаг ADO.

Например, в случае успешного соединения можно выбрать синхронный режим работы компонента:

```
procedure TForm1.ADOConnectionWillConnect(Connection: TADOConnection; var
ConnectionString, UserID, Password: WideString; var ConnectOptions:
TConnectOption; var EventStatus: TEventStatus);
begin
 if EventStatus = esOK
  then ConnectOptions := coConnectUnspecified;
end;
```

Кстати, параметр синхронности/асинхронности можно также задать при помощи свойства

```
ConnectOptions property ConnectOptions: TConnectOption;
```

После открытия соединения для выполнения собственного кода можно использовать метод-обработчик

```
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
var EventStatus: TEventStatus) of object;
property OnConnectComplete: TConnectErrorEvent;
```

Здесь, если в процессе открытия соединения возникла ошибка, параметр `EventStatus` будет равен `esErrorsOccured`, а параметр `Error` содержит объект ошибки ADO.

Теперь перейдем к вспомогательным свойствам и методам компонента `TADOConnection`, обеспечивающим соединение.

Для ограничения времени открытия соединения для медленных каналов связи служит свойство

```
property ConnectionTimeout: Integer;
```

задающее время ожидания открытия соединения в секундах. По умолчанию оно равно 15 с.

Также можно определить реакцию компонента на неиспользуемое соединение. Если через соединение не подключен ни один активный компонент, свойство

```
property KeepConnection: Boolean;
```

в значении `True` сохраняет соединение открытым. Иначе после закрытия последнего связанного компонента ADO соединение закрывается.

При необходимости провайдер соединения ADO определяется напрямую свойством

```
property Provider: WideString;
```

Имя источника данных по умолчанию задается свойством

```
property DefaultDatabase: WideString;
```

Но если этот же параметр указан в строке соединения, то он перекрывает собой значение свойства.

При необходимости прямой доступ к объекту соединения OLE DB обеспечивает свойство

```
property ConnectionObject: _Connection;
```

При открытии соединения необходимо вводить имя пользователя и его пароль. Появление стандартного диалога для этого управляется свойством

```
property LoginPrompt: Boolean;
```

Без этого диалога задать эти параметры можно с помощью свойства `ConnectionString`, метода `Open` или метода-обработчика

```
type TLoginEvent = procedure(Sender:TObject; Username, Password: string)
of object;
property OnLogin: TLoginEvent;
```

Свойство

```
type TConnectMode = (cmUnknown, cmRead, cmWrite, cmReadWrite,
cmShareDenyRead, cmShareDenyWrite, cmShareExclusive,
cmShareDenyNone);
property Mode: TConnectMode;
```

задает доступные для соединения операции:

- `cmUnknown` — разрешение неизвестно или не может быть установлено;
- `cmRead` — разрешение на чтение;
- `cmWrite` — разрешение на запись;
- `cmReadWrite` — разрешение на чтение и запись;
- `cmShareDenyRead` — разрешение на чтение для других соединений запрещено;
- `cmShareDenyWrite` — разрешение на запись для других соединений запрещено;
- `cmShareExclusive` — разрешение на открытие для других соединений запрещено;
- `cmShareDenyNone` — открытие других соединений с разрешениями запрещено.

Доступ к связанным наборам данных и командам ADO

Компонент `TADOConnection` обеспечивает доступ ко всем компонентам, которые используют его для доступа к хранилищу данных ADO. Все открытые таким образом наборы данных доступны через индексированное свойство

```
property DataSets[Index: Integer]: TCustomADODataSet;
```

Каждый элемент этого списка содержит дескриптор компонента доступа к данным ADO (тип `TCustomADODataSet`). Общее число связанных компонентов с наборами данных возвращается свойством

```
property DataSetCount: Integer;
```

Для этих компонентов тип курсора можно установить централизованно при помощи свойства

```
type TCursorLocation = (clUseServer, clUseClient);
property CursorLocation: TCursorLocation;
```

Значение `clUseClient` задает локальный курсор на стороне клиента, что позволяет выполнять любые операции с данными, в том числе не поддерживаемые сервером.

Значение `clUseServer` задает курсор на сервере, который реализует только возможности сервера, но обеспечивает быструю обработку больших массивов данных.

Например:

```
...
for i := 0 to ADOConnection.DataSetCount – 1 do
 begin
  if ADOConnection.DataSets[i].Active = True
   then ADOConnection.DataSets[i].Close;
  ADOConnection.DataSets[i].CursorLocation := clUseClient;
 end;
...
```

Помимо наборов данных, компонент `TADOConnection` обеспечивает выполнение команд ADO. Команду ADO инкапсулирует специальный компонент `TADOCommand`, который рассматривается ниже. Все команды ADO, работающие с хранилищем данных через данное соединение, доступны для управления через индексированное свойство

```
property Commands[Index: Integer]: TADOCommand
```

Каждый элемент этого списка представляет собой экземпляр класса `TADOCommand`.

Общее число доступных команд возвращается свойством

```
property CommandCount: Integer
```

Например, сразу после открытия соединения можно выполнить все связанные команды ADO, реализовав таким образом нечто вроде скрипта (листинг 23.2).

Листинг 23.2. Пример связанных команд ADO

```
procedure TForm1.ADOConnectionConnectComplete(Connection: TADOConnection;
const Error: Error; var EventStatus: TEventStatus);
var i, ErrorCnt: Integer;
begin
 if EventStatus = esOK then
 for i := 0 to ADOConnection.CommandCount – 1 do
  try
  if ADOConnection.Commands[i].CommandText <> ''
   then ADOConnection.Commands[i].Execute;
  except
   on E: Exception do Inc(ErrorCnt);
  end;
end;
```

Однако компонент `TADOConnection` может выполнять команды ADO самостоятельно, без помощи других компонентов. Для этого предусмотрен перегружаемый метод

```
function Execute(const CommandText: WideString; ExecuteOptions:
TExecuteOptions = []): _RecordSet; overload;
procedure Execute(const CommandText: WideString; var RecordsAffected:
Integer; ExecuteOptions: TExecuteOptions = [eoExecuteNoRecords]);
overload;
```

Выполнение команды осуществляется процедурой `Execute` (если команда не возвращает набор записей) или одноименной функцией `Execute` (если команда возвращает набор записей).

Параметр `CommandText` должен содержать текст команды. Параметр `RecordsAffected` возвращает число обработанных командой записей (если они есть). Параметр

```
type
 TExecuteOption = (eoAsyncExecute, eoAsyncFetch, eoAsyncFetchNonBlocking,
eoExecuteNoRecords);
 TExecuteOptions = set of TExecuteOption;
```

задает условия выполнения команды:

- `eoAsyncExecute` — команда выполняется асинхронно (соединение не будет ожидать окончания выполнения команды, а продолжит работу, обработав сигнал о завершении команды, когда он поступит);
- `eoAsyncFetch` — команда получает необходимые записи также асинхронно;
- `eoAsyncFetchNonBlocking` — команда получает необходимые записи также асинхронно, но при этом созданная нить не блокируется;
- `eoExecuteNoRecords` — команда не должна возвращать записи.

Если источник данных принял команду для выполнения и сообщил об этом соединению, вызывается метод-обработчик

```
TWillExecuteEvent = procedure(Connection: TADOConnection; var
CommandText: WideString; var CursorType: TCursorType; var LockType:
TADOLockType; var CommandType: TCommandType; var ExecuteOptions:
TExecuteOptions; var EventStatus: TEventStatus; const Command: _Command;
Recordset: _Recordset) of object;

property OnWillExecute: TWillExecuteEvent;
```

После выполнения команды вызывается метод-обработчик

```
TExecuteCompleteEvent = procedure(Connection: TADOConnection;
RecordsAffected: Integer; const Error: Error; var EventStatus:
TEventStatus; Command: _Command; const Recordset: _Recordset) of
object;

property OnExecuteComplete: TExecuteCompleteEvent;
```

Объект ошибок ADO

Все ошибки времени выполнения, возникающие при открытом соединении, сохраняются в специальном объекте ADO, инкапсулирующем коллекцию сообщений об ошибках. Доступ к объекту возможен через свойство

```
property Errors: Errors;
```

Подробнее об объекте ошибок ADO см. далее в этой главе.

Транзакции

Компонент `TADOConnection` позволяет выполнять транзакции. Методы

```
function BeginTrans: Integer;

procedure CommitTrans;

procedure RollbackTrans;
```

обеспечивают начало, фиксацию и откат транзакции соответственно. Методы-обработчики

```
TBeginTransCompleteEvent = procedure(Connection: TADOConnection;
TransactionLevel: Integer; const Error: Error; var EventStatus:
TEventStatus) of object;
```

```
property OnBeginTransComplete: TBeginTransCompleteEvent;
TConnectErrorEvent = procedure(Connection: TADOConnection; Error:
Error; var EventStatus: TEventStatus) of object;
property OnCommitTransComplete: TConnectErrorEvent;
```

вызываются после начала и фиксации транзакции.

Свойство

```
type TIsolationLevel = (ilUnspecified, ilChaos, ilReadUncommitted,
ilBrowse, ilCursorStability, ilReadCommitted, ilRepeatableRead,
ilSerializable, ilIsolated);
property IsolationLevel: TIsolationLevel;
```

позволяет задать уровень изоляции транзакции:

- `ilUnspecified` — уровень изоляции не задается;
- `ilChaos` — изменения более защищенных транзакций не перезаписываются данной;
- `ilReadUncommitted` — незафиксированные изменения других транзакций видимы;
- `ilBrowse` — незафиксированные изменения других транзакций видимы;
- `ilCursorStability` — изменения других транзакций видимы только после фиксации;
- `ilReadCommitted` — изменения других транзакций видимы только после фиксации;
- `ilRepeatableRead` — изменения других транзакций не видимы, но доступны при обновлении данных;
- `ilSerializable` — транзакция выполняется изолированно от других транзакций;
- `ilIsolated` — транзакция выполняется изолированно от других транзакций.

Свойство

```
TXactAttribute = (xaCommitRetaining, xaAbortRetaining);
property Attributes: TXactAttributes;
```

задает способ управления транзакциями при их фиксации и откате:

- `xaCommitRetaining` — после фиксации очередной транзакции автоматически начинается выполнение новой;
- `xaAbortRetaining` — после отката очередной транзакции автоматически начинается выполнение новой.

Наборы данных ADO

В категории **dbGo** Палитры инструментов Delphi кроме компонентов соединения есть стандартные компоненты, инкапсулирующие набор данных и адаптированные для работы с хранилищем данных ADO (рис. 23.7).

Это компоненты

- `TADODataSet` — универсальный набор данных;
- `TADOTable` — таблица БД;
- `TADOQuery` — запрос SQL;
- `TADOStoredProc` — хранимая процедура.

Как и положено для компонентов, инкапсулирующих набор данных, их общим предком является класс `TDataSet`, предоставляющий базовые функции управления набором данных *(см. главу 16)*.

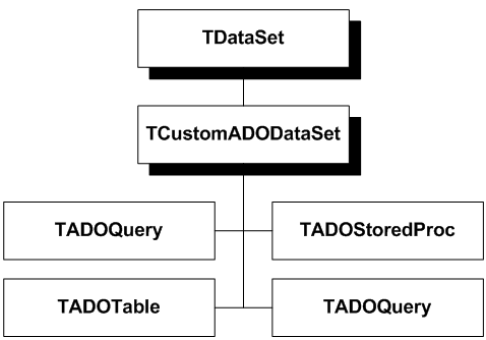


Рис. 23.7. Иерархия классов наборов данных ADO

Компоненты ADO обладают обычным набором свойств и методов, а необходимый для доступа к данным через ADO механизм наследуют от своего общего предка — класса `TCustomADODataSet`. Кроме этого, класс `TCustomADODataSet` содержит ряд общих для всех потомков свойств и методов, рассмотреть которые будет очень полезно. Поэтому сначала мы изучим класс `TCustomADODataSet`, и только потом перейдем к компонентам ADO.

Класс *TCustomADODataSet*

Класс `TCustomADODataSet` инкапсулирует механизм доступа к хранилищу данных через ADO. Этот класс наполняет абстрактные методы общего предка `TDataSet` функциями конкретного механизма доступа к данным.

Поэтому здесь мы рассмотрим только уникальные свойства и методы класса `TCustomADODataSet`, обеспечивающие работу с ADO.

Соединение набора данных с хранилищем данных ADO осуществляется через компонент `TADOConnection` (свойство `Connection`) или путем задания параметров соединения через свойство `ConnectionString`.

Набор данных

Перед открытием набора данных необходимо установить тип используемой при редактировании записей блокировки. Для этого применяется свойство

```
type TADOLockType = (ltUnspecified, ltReadOnly, ltPessimistic,
ltOptimistic, ltBatchOptimistic);
property LockType: TADOLockType;
```

Здесь

- `ltUnspecified` — блокировка задается источником данных, а не компонентом;
- `ltReadOnly` — набор данных откроется в режиме "только для чтения";
- `ltPessimistic` — редактируемая запись блокируется на все время редактирования до момента сохранения в хранилище данных;
- `ltOptimistic` — запись блокируется только на время сохранения изменений в хранилище данных;
- `ltBatchOptimistic` — запись блокируется на время сохранения в хранилище данных при вызове метода `UpdateBatch`.

> **Примечание**
>
> Для того чтобы установка блокировки сработала, свойство `LockType` должно быть обязательно модифицировано до открытия набора данных.

Набор данных открывается методом `Open` и закрывается методом `Close`. Также допустимо свойство

```
property Active: Boolean;
```

Текущее состояние набора данных можно определить свойством

```
type
  TObjectState = (stClosed, stOpen, stConnecting, stExecuting,
stFetching);
  TObjectStates = set of TObjectState;
property RecordsetState: TObjectStates;
```

Набор данных в компонентах ADO основан на реализации объекта набора записей ADO, прямой доступ к этому объекту возможен при помощи свойства

```
property Recordset: _Recordset;
```

Но так как все основные методы интерфейсов объекта набора записей ADO перекрыты методами класса, в обычных случаях прямой доступ к объекту вам не понадобится.

После обновления набора данных вызывается метод-обработчик

```
TRecordsetEvent = procedure(DataSet: TCustomADODataSet; const Error:
Error; var EventStatus: TEventStatus) of object;
property OnFetchComplete: TRecordsetEvent;
```

где `Error` — ссылка на объект ошибки ADO, если она возникла.

Если же набор данных работает в асинхронном режиме, при обновлении вызывается метод-обработчик

```
TFetchProgressEvent = procedure(DataSet: TCustomADODataSet; Progress,
MaxProgress: Integer; var EventStatus: TEventStatus) of object;
property OnFetchProgress: TFetchProgressEvent;
```

где параметр `Progress` показывает долю выполнения операции.

Курсор набора данных

Для набора данных ADO в зависимости от его назначения можно выбрать тип и местоположение курсора.

Местоположение курсора задается свойством

```
type TCursorLocation = (clUseServer, clUseClient);
property CursorLocation: TCursorLocation;
```

Курсор может находиться на сервере (`clUseServer`) или на клиенте (`clUseClient`).

- Серверный курсор используется при работе с большими наборами данных, которые нецелесообразно пересылать клиенту целиком. При этом несколько снижается скорость работы клиентского набора данных.
- Клиентский курсор обеспечивает передачу набора данных клиенту. Это положительно сказывается на быстродействии, но такой курсор целесообразен только для небольших наборов данных, не загружающих канал связи с сервером.

Для клиентского курсора необходимо дополнительно установить свойство

```
TMarshalOption = (moMarshalAll, moMarshalModifiedOnly);
property MarshalOptions: TmarshalOption
```

которое управляет обменом данных клиента с сервером. Если соединение с сервером быстрое, можно выбрать значение `moMarshalAll`, разрешающее возврат серверу всех записей набора данных. В противном случае для ускорения работы компонента можно применить свойство `moMarshalModifiedOnly`, обеспечивающее возврат только модифицированных клиентом записей.

Тип курсора определяется свойством

```
TCursorType = (ctUnspecified, ctOpenForwardOnly, ctKeyset,
ctDynamic, ctStatic);
property CursorType: TCursorType;
```

Здесь

- `ctUnspecified` — курсор не задан, тип курсора определяется возможностями источника данных;
- `ctOpenForwardOnly` — однонаправленный курсор, допускающий перемещение только вперед; используется при необходимости быстрого одиночного прохода по всем записям набора данных;
- `ctKeyset` — двунаправленный локальный курсор, не обеспечивающий просмотр добавленных и удаленных другими пользователями записей;
- `ctDynamic` — двунаправленный курсор, отображает все изменения, требует наибольших затрат ресурсов;
- `ctStatic` — двунаправленный курсор, полностью игнорирует изменения, внесенные другими пользователями.

Примечание

Если курсор расположен на клиенте (`CursorType = clUseClient`), то для него доступен только один тип — `ctStatic`.

Соответственно до и после каждого перемещения курсора в наборе данных вызываются методы-обработчики:

```
TRecordsetReasonEvent = procedure(DataSet: TCustomADODataSet; const
Reason: TEventReason; var EventStatus: TEventStatus) of object;
property OnWillMove: TRecordsetReasonEvent;
```

и

```
TRecordsetErrorEvent = procedure(DataSet: TCustomADODataSet; const
Reason: TEventReason; const Error: Error; var EventStatus: TEventStatus)
of object;
property OnMoveComplete: TRecordsetErrorEvent;
```

где параметр `Reason` позволяет узнать, какой метод вызвал это перемещение.

Локальный буфер

После передачи клиенту записи набора данных размещаются в локальном буфере, размер которого определяется свойством

```
property CacheSize: Integer;
```

Значение свойства есть число записей, помещаемых в локальный буфер, и оно не может быть меньше 1. Очевидно, что при достаточно большом раз-

мере буфера компонент будет обращаться к источнику данных не так часто, но при этом большой буфер заметно замедлит открытие набора данных.

Кроме этого, при выборе размера локального буфера необходимо учитывать доступную компоненту память. Это можно сделать путем несложных вычислений:

```
CacheSizeInMem := ADODataSet.CacheSize * ADODataSet.RecordSize;
```

где `RecordSize` — свойство `property RecordSize: Word;`

возвращающее размер одной записи в байтах.

Как видите, компоненты ADO не избежали общей проблемы клиентских данных: при плохом качестве соединения работа приложения замедляется. Однако кое-что все-таки сделать можно. Если при навигации по записям вам не требуется отображать данные в визуальных компонентах пользовательского интерфейса, свойство

```
property BlockReadSize: Integer;
```

позволяет организовать блочную пересылку данных. Оно задает число записей, помещаемых в один блок. При этом набор данных переходит в состояние `dsBlockRead`. По умолчанию блочная пересылка не используется и значение свойства равно 0.

Также можно ограничить максимальный размер набора данных. Свойство

```
property MaxRecords: Integer
```

задает максимальное число записей набора данных. По умолчанию свойство имеет значение 0 и набор данных не ограничен. А общее число записей набора данных на данный момент возвращает свойство только для чтения

```
property RecordCount: Integer;
```

При достижении последней записи набора данных вызывается метод-обработчик

```
TEndOfRecordsetEvent = procedure (DataSet: TCustomADODataSet; var
MoreData: WordBool; var EventStatus: TEventStatus) of object;
property OnEndOfRecordset: TEndOfRecordsetEvent;
```

При этом параметр `MoreData` показывает, действительно ли достигнут конец набора данных. Если `MoreData = True`, то это означает, что в хранилище данных еще имеются записи, не переданные клиенту.

Состояние записи

Класс `TCustomADODataSet` обладает дополнительными возможностями, которые позволяют отслеживать состояние каждой записи.

Для текущей записи набора данных можно определить ее состояние. Для этого предназначено свойство

```
TRecordStatus = (rsOK, rsNew, rsModified, rsDeleted, rsUnmodified,
rsInvalid,    rsMultipleChanges,    rsPendingChanges,    rsCanceled,
rsCantRelease,     rsConcurrencyViolation,     rsIntegrityViolation,
rsMaxChangesExceeded,          rsObjectOpen,          rsOutOfMemory,
rsPermissionDenied, rsSchemaViolation, rsDBDeleted);
property RecordStatus: TRecordStatusSet;
```

Здесь параметры имеют следующий смысл:

- `rsOK` — запись успешно сохранена;
- `rsNew` — запись добавлена;
- `rsModified` — запись была изменена;
- `rsDeleted` — запись удалена;
- `rsUnmodified` — запись без изменений;
- `rsInvalid` — запись не может быть сохранена из-за неверной закладки;
- `rsMultipleChanges` — запись не может быть сохранена из-за множественных изменений;
- `rsPendingChanges` — запись не может быть сохранена из-за ссылки на несохраненные изменения;
- `rsCanceled` — операция с записью была отменена;
- `rsCantRelease` — запись заблокирована;
- `rsConcurrencyViolation` — запись не может быть сохранена из-за типа блокировки;
- `rsIntegrityViolation` — нарушена ссылочная целостность;
- `rsMaxChangesExceeded` — слишком много изменений;
- `rsObjectOpen` — конфликт с объектом базы данных;
- `rsOutOfMemory` — недостаток памяти,
- `rsPermissionDenied` — нет доступа;
- `rsSchemaViolation` — нарушение структуры данных;
- `rsDBDeleted` — запись удалена в БД.

Как видите, благодаря этому свойству состояние отдельной записи может быть определено очень точно.

Кроме этого, метод

```
type
  TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
function UpdateStatus: TUpdateStatus; override;
```

возвращает информацию о состоянии текущей записи.

Соответственно до и после изменения записи вызываются методы-обработчики

```
TWillChangeRecordEvent = procedure(DataSet: TCustomADODataSet; const
Reason: TEventReason; const RecordCount: Integer; var EventStatus:
TEventStatus) of object;
property OnWillChangeRecord: TWillChangeRecordEvent;
```

и

```
TRecordChangeCompleteEvent = procedure(DataSet: TCustomADODataSet; const
Reason: TEventReason; const RecordCount: Integer; const Error: Error; var
EventStatus: TEventStatus) of object;
property OnRecordChangeComplete: TrecordChangeCompleteEvent;
```

где параметр `Reason` позволяет узнать, какой метод изменил записи, а `RecordCount` возвращает число измененных записей.

Фильтрация

Помимо обычной фильтрации, основанной на свойствах `Filter`, `Filtered` и методе-обработчике `OnFilterRecord`, класс `TCustomADODataSet` предоставляет разработчику дополнительные возможности.

Свойство

```
TFilterGroup = (fgUnassigned, fgNone, fgPendingRecords,
fgAffectedRecords, fgFetchedRecords, fgPredicate, fgConflictingRecords);
property FilterGroup: TFilterGroup;
```

задает групповой фильтр для записей, основываясь на информации о состоянии каждой записи набора данных, подобно рассмотренному ранее свойству `RecordStatus`.

Фильтрация возможна по следующим параметрам:

- `fgUnassigned` — фильтр не задан;
- `fgNone` — все ограничения, заданные фильтром, снимаются, отображаются все записи набора данных;
- `fgPendingRecords` — отображаются измененные записи, не сохраненные в хранилище данных при вызове методов `UpdateBatch` или `CancelBatch`;
- `fgAffectedRecords` — отображаются записи, обработанные при последнем сохранении в хранилище данных;
- `fgFetchedRecords` — отображает записи, полученные при последнем обновлении из источника данных;
- `fgPredicate` — отображает только удаленные записи;
- `fgConflictingRecords` — отображает модифицированные записи, при сохранении которых в хранилище данных возникла ошибка.

Для того чтобы групповая фильтрация заработала, необходимы два дополнительных условия. Во-первых, фильтрация должна быть включена — свойство `Filtered` должно иметь значение `True`. Во-вторых, свойство `LockType` должно иметь значение `ltBatchOptimistic` (листинг 23.3).

Листинг 23.3. Пример установок групповой фильтрации

```
with ADODataSet do
 begin
  Close;
  LockType := ltbatchOptimistic;
  Filtered := True;
  FilterGroup := fgFetchedRecords;
  Open;
 end;
```

Метод

```
procedure FilterOnBookmarks(Bookmarks: array of const);
```

включает фильтрацию по существующим закладкам. Для этого предварительно необходимо при помощи метода `GetBookmark` установить закладки на интересующих записях. При вызове метода `FilterOnBookmarks` автоматически очищается свойство `Filter`, а свойству `FilterGroup` присваивается значение `gUnassigned`.

Поиск

Быстрый и гибкий поиск по полям текущего индекса набора данных обеспечивает метод

```
SeekOption = (soFirstEQ, soLastEQ, soAfterEQ, soAfter, soBeforeEQ,
soBefore);
function Seek(const KeyValues: Variant; SeekOption: TSeekOption =
soFirstEQ): Boolean;
```

В параметре `KeyValues` должны быть перечислены необходимые значения полей индекса. Параметр `SeekOption` управляет процессом поиска:

- `soFirstEQ` — курсор устанавливается на первую найденную запись;
- `soLastEQ` — курсор устанавливается на последнюю найденную запись;
- `soAfterEQ` — курсор устанавливается на найденную запись или, если запись не найдена, сразу после того места, где она могла находиться;
- `soAfter` — курсор устанавливается сразу после найденной записи;

- `soBeforeEQ` — курсор устанавливается на найденную запись или, если запись не найдена, перед тем местом, где она могла находиться;
- `soBefore` — курсор устанавливается перед найденной записью.

Сортировка

Свойство

```
property Sort: WideString;
```

предоставляет простой способ сортировки по произвольному сочетанию полей. Оно должно включать через запятую имена нужных полей и признак прямого или обратного порядка сортировки:

```
ADODataSet.Sort := 'FirstField DESC';
```

Если порядок сортировки не указан, по умолчанию задается прямой порядок.

Команда ADO

Для выполнения запросов к источнику данных любой компонент ADO инкапсулирует специальный объект команды ADO.

При использовании компонентов-потомков класса `TCustomADODataSet` обычно нет необходимости применять объект команды напрямую. И хотя все реальное взаимодействие объекта набора данных ADO с источником данных осуществляется через объект команды, настройка и выполнение команды скрыты в свойствах и методах компонентов ADO. Тем не менее, в классе `TCustomADODataSet` доступ к объекту команды можно получить при помощи свойства

```
property Command: TADOCommand;
```

Примечание

При необходимости выполнить команду ADO, не связанную с конкретным набором данных, разработчик может использовать отдельный компонент `TADOCommand`, так же расположенный в категории **dbGo** Палитры инструментов.

Тип команды задается свойством

```
type
  TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc, cmdFile,
cmdTableDirect);
property CommandType: TCommandType;
```

Здесь

- `cmdUnknown` — тип команды не известен и будет определен источником данных;

- `cmdText` — текстовая команда, интерпретируемая источником данных (например, запрос SQL); текст должен быть составлен с учетом правил для конкретного источника данных;
- `cmdTable` — команда на получение набора данных таблицы из хранилища данных;
- `cmdStoredProc` — команда на выполнение хранимой процедуры;
- `cmdFile` — команда на получение набора данных, сохраненного в файле в формате, используемом конкретным источником данных;
- `cmdTableDirect` — команда на получение набора данных таблицы напрямую, например, из файла таблицы.

Текст команды, представленный свойством

```
property CommandText: WideString;
```

обязательно должен быть согласован с ее типом.

Для ограничения времени ожидания выполнения команды имеется свойство

```
property CommandTimeout: Integer;
```

В компонентах наборов данных ADO команды выполняется при выполнении следующих операций:

- открытие и закрытие набора данных;
- выполнение запросов и хранимых процедур;
- обновление набора данных;
- сохранение изменений;
- групповые операции.

Разработчик может повлиять на способ выполнения команды. Для этого он может изменить свойство

```
type
  TExecuteOption = (eoAsyncExecute, eoAsyncFetch,
  eoAsyncFetchNonBlocking, eoExecuteNoRecords);
  TExecuteOptions = set of TExecuteOption;
property ExecuteOptions: TExecuteOptions;
```

Здесь:

- `eoAsyncExecute` — асинхронное выполнение команды;
- `eoAsyncFetch` — асинхронное выполнение команды на обновление набора данных;
- `eoAsyncFetchNonBlocking` — асинхронное выполнение команды на обновление набора данных без установки блокировки;
- `eoExecuteNoRecords` — выполнение команды не требует возвращения набора данных.

Групповые операции

Как уже говорилось, наборы данных ADO используют на клиентской стороне локальный кэш для хранения данных и сделанных изменений. Благодаря этому и появилась возможность реализовать групповые операции. В данном режиме все сделанные изменения не передаются немедленно источнику данных, а накапливаются в локальном кэше. Это повышает скорость работы и позволяет сохранять сразу группу модифицированных записей.

Из отрицательных сторон этого метода стоит отметить, что, пока изменения находятся на клиенте, они недоступны другим пользователям. В данной ситуации могут возникать потери данных.

Для перевода набора данных в режим групповых операций необходимо выполнить следующие условия.

Набор данных должен использовать клиентский курсор:

```
ADODataSet.CursorLocation := clUseClient;
```

Курсор должен иметь тип `ctStatic`:

```
ADODataSet.CursorType := ctStatic;
```

Блокировка должна иметь значение `ltBatchOptimistic`:

```
ADODataSet.LockType := ltBatchOptimistic;
```

Для передачи сделанных изменений в хранилище данных в компонентах ADO используется метод

```
procedure UpdateBatch(AffectRecords: TAffectRecords = arAll);
```

Для отмены всех сделанных, но не сохраненных методом `UpdateBatch` изменений служит метод

```
procedure CancelBatch(AffectRecords: TAffectRecords = arAll);
```

Параметр `TAffectRecords` позволяет задать тип записей, к которым применяется данная операция:

```
TAffectRecords = (arCurrent, arFiltered, arAll, arAllChapters);
```

- `arCurrent` — операция выполняется только для текущей записи;
- `arFiltered` — операция выполняется для записей из работающего фильтра;
- `rAll` — операция выполняется для всех записей;
- `arAllChapters` — операция выполняется для всех записей текущего набора данных (включая невидимые из-за включенного фильтра), а также для всех вложенных наборов данных.

Параметры

Многие компоненты ADO, инкапсулирующие набор записей, должны обеспечивать применение параметров запросов. Для этого в них предусмотрен специальный класс `TParameters`.

Для каждого параметра из коллекции класса `TParameters` создается отдельный класс `TParameter`.

Этот класс является наследником класса коллекции `TCollection` и инкапсулирует индексированный список отдельных параметров. Напомним, что для работы с параметрами обычных запросов в компонентах запросов и хранимых процедур используется класс `TParams` (например, в компонентах dbExpress), также происходящий от класса коллекции.

Методы этих двух классов совпадают, а свойства имеют некоторые отличия. Для представления параметров команд в ADO имеется специальный объект параметров, который активно применяется в процессе работы компонентов ADO, инкапсулирующих набор данных.

Поэтому для компонентов dbGo в VCL был создан собственный класс параметров.

Класс *TParameters*

Главное, для чего предназначен класс `TParameters` — содержать список параметров. Индексированный список параметров представлен свойством

```
property Items[Index: Integer]: TParameter;
```

Текущие значения параметров можно получить из индексированного свойства

```
property ParamValues[const ParamName: String]: Variant;
```

При этом доступ к конкретному значению осуществляется по имени параметра:

```
Edit1.Text := ADODataSet.Parameters.ParamValues['ParamOne'];
```

Список параметров можно обновлять при помощи методов

```
function AddParameter: TParameter;
```

и

```
function CreateParameter(const Name: WideString; DataType:
TDataType; Direction: TParameterDirection; Size: Integer; Value:
OleVariant): TParameter;
```

Первый метод просто создает новый объект параметра и добавляет его к списку. Затем необходимо задать все свойства нового параметра:

```
var NewParam: TParameter;

...

 NewParam := ADODataSet.Parameters.AddParameter;
```

```
NewParam.Name := 'ParamTwo';
NewParam.DataType := ftInteger;
NewParam.Direction := pdInput;
NewParam.Value := 0;
```

Метод `CreateParameter` создает новый параметр и определяет его свойства:

- `Name` — имя параметра;
- `DataType` — тип данных параметра, соответствующий типу поля таблицы БД (тип `TFieldType`).
- `Direction` — тип параметра; помимо стандартных (`dUnknown`, `pdInput`, `pdOutput`, `pdInputOutput`), тип `TParameterDirection` имеет дополнительный тип `pdReturnValue`, определяющий любое возвращаемое значение;
- `Size` — максимальный размер значения параметра;
- `Value` — значение параметра.

При работе с параметрами полезно вызывать их по именам, а не по абсолютным индексам в списке. Для этого есть метод

```
function ParamByName(const Value: WideString ): TParameter;
```

Список параметров всегда должен соответствовать запросу или хранимой процедуре. Для обновления списка существует метод

```
procedure Refresh;
```

Также вы можете создать список параметров для не связанного с данным объектом параметров запроса. Для этого используется метод

```
function ParseSQL(SQL: String; DoCreate: Boolean): String;
```

где `DoCreate` определяет, удалять ли перед анализом существующие параметры.

Класс *TParameter*

Класс `TParameter` инкапсулирует отдельный параметр.

Имя параметра определяется свойством

```
property Name: WideString;
```

Тип данных, которому должно соответствовать его значение, задается свойством

```
TDataType = TFieldType;
property DataType: TDataType;
```

И так как параметры взаимодействуют с полями таблиц БД, то тип данных параметров совпадает с типами данных полей.

От типа данных зависит размер параметра

```
property Size: Integer;
```

который может быть изменен для строкового или символьного типа данных и им подобных.

Само значение параметра содержится в свойстве

```
property Value: OleVariant;
```

А свойство

```
type
  TParameterAttribute = (paSigned, paNullable, paLong);
  TParameterAttributes = set of TParameterAttribute;
property Attributes: TParameterAttributes;
```

контролирует значение, присваиваемое параметру:

- `paSigned` — значение может быть символьным;
- `paNullable` — значение параметра может быть пустым;
- `paLong` — значение может содержать данные типа BLOB.

Тип параметра задается свойством

```
type TParameterDirection = (pdUnknown, pdInput, pdOutput, pdInputOutput,
pdReturnValue);
property Direction: TParameterDirection;
```

Здесь:

- `pdUnknown` — неизвестный тип, источник данных попытается определить его самостоятельно;
- `pdInput` — входной параметр, используется в запросах и хранимых процедурах;
- `pdOutput` — выходной параметр, используется в хранимых процедурах;
- `pdInputOutput` — входной и выходной параметр одновременно, применяется в хранимых процедурах;
- `pdReturnValue` — параметр для передачи значения.

Если параметр должен передавать большие бинарные массивы (например, изображения или файлы), то значение параметра можно загрузить методами

```
procedure LoadFromFile(const FileName: String; DataType: TDataType);
```

и

```
procedure LoadFromStream(Stream: TStream; DataType: TDataType);
```

Компонент *TADODataSet*

Компонент `TADODataSet` предназначен для представления набора данных из хранилища данных ADO. Он прост в применении, имея несколько собственных свойств и методов, и использует функции своего предка — класса `TCustomADODataSet`.

Это единственный компонент dbGo, инкапсулирующий набор данных, для которого опубликованы свойства, позволяющие управлять командой ADO. Это свойства

```
property CommandText: WideString;
```

и

```
property CommandType: TCommandType;
```

В результате компонент представляет собой гибкий инструмент, который позволяет (в зависимости от типа команды и ее текста) получать данные из таблиц, запросов SQL, хранимых процедур, файлов и т. д. Например, вы выбираете нужное значение свойства `CommandType = cmdText` и заносите в свойство `CommandText` текст запроса SQL из редактора:

```
ADODataSet.CommandType = cmdText;
ADODataSet.CommandText := Memo1.Lines.Text;
```

и запрос SQL готов к выполнению.

Примечание

Для запросов SQL можно применять только язык Data Manipulation Language (использовать только SELECT).

Соединение с базой данных задается свойствами `ConnectionString` или `Connection`.

Набор данных открывается и закрывается свойством `Active` или методами `Open` и `Close`.

В приложениях компонент можно применять как все обычные компоненты доступа к данным, связывая инкапсулированный в нем набор данных с визуальными компонентами отображения данных через компонент `TDataSource`.

Компонент *TADOTable*

Компонент `TADOTable` обеспечивает реализацию в приложениях Delphi таблиц БД, подключенных через провайдеры OLE DB. По своим функциональным возможностям и применению он подобен стандартному табличному компоненту *(см. главу 16)*.

Как вы уже знаете, в основе компонента лежит команда ADO, но ее свойства настроены заранее и изменению не подлежат.

Имя таблицы БД задается свойством

```
property TableName: WideString;
```

Другие свойства и методы компонента обеспечивают применение индексов (этой возможности лишен любой компонент запроса).

Так как не все провайдеры ADO обеспечивают прямое использование таблиц БД, то для доступа к ним может понадобиться запрос SQL. Если свойство

```
property TableDirect: Boolean;
```

имеет значение `True`, осуществляется прямой доступ к таблице. В противном случае компонент генерирует соответствующий запрос.

Свойство

```
property ReadOnly: Boolean;
```

Позволяет включить или отключить для таблицы режим "только для чтения".

Компонент *TADOQuery*

Компонент `TADOQuery` обеспечивает применение запросов SQL при работе с данными через ADO. По своей функциональности он подобен стандартному компоненту запроса *(см. главу 16)*.

Текст запроса задается свойством

```
property SQL: TStrings;
```

Параметры запроса определяются свойством

```
property Parameters: TParameters;
```

Если запрос должен возвращать набор данных, для его открытия необходимо свойство

```
property Active: Boolean;
```

или метод

```
procedure Open;
```

В противном случае достаточно использовать метод

```
function ExecSQL: Integer; ExecSQL
```

Число обработанных запросом записей возвращает свойство

```
property RowsAffected: Integer;
```

Компонент *TADOStoredProc*

Компонент `TADOStoredProc` позволяет реализовать в приложениях Delphi, обращающихся к данным через ADO, хранимые процедуры. Он подобен стандартному компоненту хранимой процедуры *(см. главу 16)*.

Имя хранимой процедуры определяется свойством

```
property ProcedureName: WideString;
```

Для определения входных и выходных параметров служит свойство

```
property Parameters: TParameters;
```

Если процедура будет использоваться без изменений многократно, имеет смысл заранее подготовить ее выполнение на сервере. Для этого свойству

```
property Prepared: Boolean;
```

присваивается значение `True`.

Команды ADO

Команде ADO, которой мы уделяли так много внимания в этой главе, в VCL Delphi соответствует компонент `TADOCommand`. Методы этого компонента во многом совпадают с методами класса `TCustomADODataSet`, хотя этот класс не является предком компонента. Он предназначен для выполнения команд, которые не возвращают наборы данных.

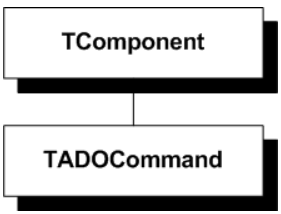


Рис. 23.8. Иерархия классов компонента `TADOCommand`

Так как компоненту `TADOCommand` нет необходимости обеспечивать работу набора записей, его непосредственным предком является класс `TComponent` (рис. 23.8). К его функциональности просто добавлен механизм соединения с БД через ADO и средства представления команды.

Команда передается в хранилище данных ADO через собственное соединение или через компонент `TADOConnection`, аналогично другим компонентам dbGo.

Текстовое представление выполняемой команды должно содержаться в свойстве

```
property CommandText: WideString;
```

однако команду можно задать и другим способом. Прямая ссылка на нужный объект команды ADO может быть задана свойством

```
property CommandObject: _Command;
```

Тип команды определяется свойством

```
type TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc,
cmdFile, cmdTableDirect);
property CommandType: TCommandType;
```

Так как тип `TCommandType` также используется в классе `TCustomADODataSet`, где необходимо представлять все возможные виды команд, по отношению к компоненту `TADOCommand` этот тип обладает избыточностью. Здесь нельзя установить значения `cmdTable`, `cmdFile` и `cmdTableDirect`, а тип `cmdStoredProc` должен обозначать только те хранимые процедуры, которые не возвращают набор данных.

Если команда содержит текст запроса SQL, свойство `CommandType` должно иметь значение `cmdText`.

Для вызова хранимой процедуры необходимо задать тип `cmdStoredProc`, а в свойстве `CommandText` ввести имя процедуры.

Если для выполнения команды необходимо задать параметры, используется свойство

```
property Parameters: TParameters;
```

Выполнение команды осуществляется методом `Execute`:

```
function Execute: _RecordSet; overload;
function Execute(const Parameters: OleVariant): _Recordset;
overload;
function Execute(var RecordsAffected: Integer; var Parameters:
OleVariant; ExecuteOptions: TExecuteOptions = []): _RecordSet;
overload;
```

Разработчик может выбрать любую из представленных нотаций перегружаемого метода.

- Параметр `RecordsAffected` возвращает число обработанных записей.
- Параметр `Parameters` задает параметры команды.
- Параметр `ExecuteOptions` определяет условия выполнения команды:

```
TExecuteOption = (eoAsyncExecute, eoAsyncFetch,
eoAsyncFetchNonBlocking, eoExecuteNoRecords);
  TExecuteOptions = set of TExecuteOption;
```

- `eoAsyncExecute` — асинхронное выполнение команды;
- `eoAsyncFetch` — асинхронная передача данных;
- `eoAsyncFetchNonBlocking` — асинхронная передача данных без блокирования потока;
- `eoExecuteNoRecords` — если команда возвращает набор записей, то они не передаются в компонент.

При работе с компонентом `TADOConnection` желательно установить опцию `eoExecuteNoRecords`.

Для прерывания выполнения команды используется метод

```
procedure Cancel;
```

Текущее состояние команды можно определить свойством

```
type
  TObjectState = (stClosed, stOpen, stConnecting, stExecuting,
stFetching);
  TObjectStates = set of TObjectState;
property States: TObjectStates;
```

Объект ошибок ADO

При рассказе о компонентах ADO в этой главе мы довольно часто упоминали об объектах ошибок ADO. Эти объекты содержат информацию об ошибке, возникшей при выполнении операции каким-либо объектом ADO.

В Delphi для объекта ошибки не предусмотрен специальный тип, но разработчик может использовать его методы интерфейса `Error`, предоставляемого многими методами других объектов ADO. Например тип

```
TRecordsetEvent = procedure(DataSet: TCustomADODataSet; const Error:
Error; var EventStatus: TEventStatus) of object;
```

метода-обработчика, вызываемого после обновления набора данных, содержит параметр `Error`, дающий нам искомую ссылку.

Рассмотрим полезные свойства объекта ошибок ADO.

Свойство

```
property Description: WideString read Get_Description;
```

возвращает описание ошибки, переданное из объекта, в котором ошибка произошла.

Свойство

```
property SQLState: WideString read Get_SQLState;
```

содержит текст команды, вызвавшей ошибку.

Свойство

```
property NativeError: Integer read Get_NativeError;
```

возвращает код ошибки, переданный из объекта, в котором ошибка произошла.

Пример приложения ADO

Теперь попробуем применить на практике представленную в этой главе информацию о реализации ADO в Delphi. В качестве примера создадим простое приложение ADO Demo, которое умеет отображать пару таблиц БД, сохранять изменения при помощи групповых операций, сортировать записи и устанавливать фильтры на выбранные записи (рис. 23.9).

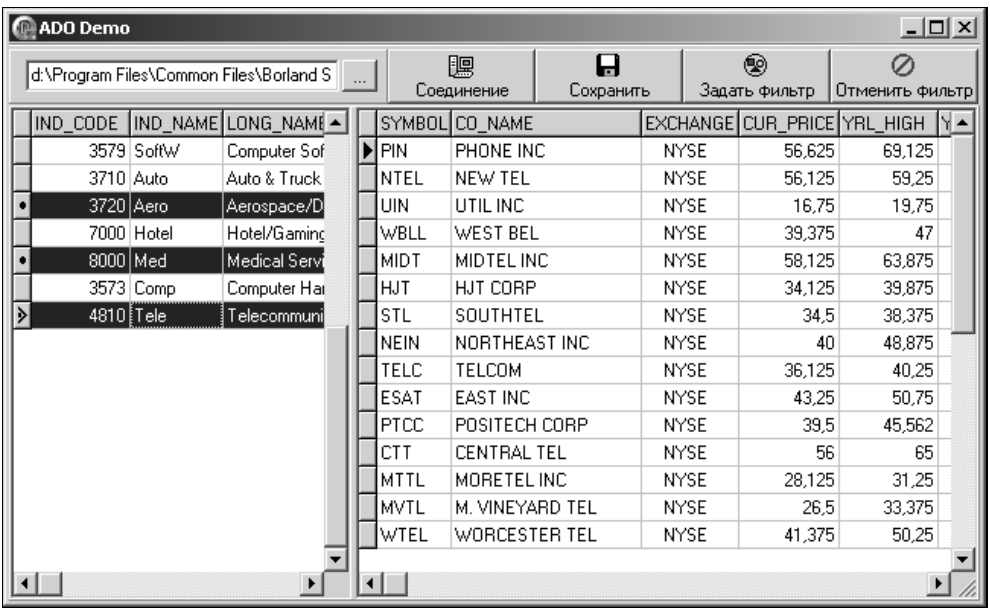


Рис. 23.9. Главное окно приложения ADO Demo

В качестве источника данных выберем файлы dBase, имеющиеся в демонстрационной базе данных Delphi ..\Program Files\Common Files\Borland Shared\Data. Для приложения выберем две таблицы: `INDUSTRY` и `MASTER`. Они связаны между собой внешним ключом по полям `IND_CODE` и `INDUSTRY` соответственно.

Примечание

При запуске примера на вашем компьютере на параметры соединения ADO может повлиять локализация операционной системы. Поэтому рекомендуется просто обновить заданные в проекте параметры соединения в компоненте `ADOConn`.

Таблицу `INDUSTRY` можно редактировать, она инкапсулирована в компоненте `tblIndustry` типа `TADOTable` и отображается в левом компоненте `TDBGrid`. А таблица `MASTER` инкапсулирована в компоненте `tblMaster` и предназначена только для просмотра. Эти два компонента связаны отношением "один ко многим" при помощи свойств `MasterSource` и `MasterFields`.

Листинг 23.4 содержит код секции `implementation` модуля `uMain` приложения ADO Demo.

Листинг 23.4. Секция `implementation` модуля `uMain` приложения ADO Demo

```
implementation
uses IniFiles, FileCtrl;
const sIniFileName: String = 'ADODemo.ini';
      sEmptyDefDB:  String = 'Database path is empty';
      sEmptyFilter: String = 'Records for filter is not selected';
{$R *.dfm}
procedure TfmMain.FormShow(Sender: TObject);
begin
 with TIniFile.Create(sIniFileName) do
 try
  DefDBStr := ReaDString('DefDB', 'DefDBStr', '');
  edDefDB.Text := DefDBStr;
 finally
  Free;
 end;
 SetLength(Bookmarks, 0);
end;
procedure TfmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 with TIniFile.Create(sIniFileName) do
 try
  WriteString('DefDB', 'DefDBStr', edDefDB.Text);
```

```
 finally
  Free;
 end;
end;

procedure TfmMain.sbDefDBClick(Sender: TObject);
begin
 if SelectDirectory(DefDBStr, [], 0)
  then edDefDB.Text := DefDBStr;
end;
procedure TfmMain.tbConnectClick(Sender: TObject);
begin
 ADOConn.Close;
 ADOConn.DefaultDatabase := '';
 if DefDBStr = '' then
 begin
  MessageDlg(sEmptyDefDB, mtError, [mbOK], 0);
  Abort;
 end
 else
 begin
  ADOConn.DefaultDatabase := DefDBStr;
  ADOConn.Open;
 end;
end;
procedure TfmMain.tbSaveClick(Sender: TObject);
begin
 tblIndustry.UpdateBatch();
end;
procedure TfmMain.tbFilterClick(Sender: TObject);
var i: Integer;
begin
 if dbgIndustry.SelectedRows.Count > 0 then
 begin
  SetLength(Bookmarks, dbgIndustry.SelectedRows.Count);
  for i := 0 to dbgIndustry.SelectedRows.Count - 1 do
```

```
  begin
   Bookmarks[i].VType := vtPointer;
   Bookmarks[i].VPointer := pointer(dbgIndustry.SelectedRows[i]);
  end;
  tblIndustry.FilterOnBookmarks(Bookmarks);
 end
 else
  MessageDlg(sEmptyFilter, mtWarning, [mbOK], 0);
end;
procedure TfmMain.tbUnFilterClick(Sender: TObject);
begin
 tblIndustry.Filtered := False;
 dbgIndustry.SelectedRows.Clear;
end;
procedure TfmMain.dbgIndustryTitleClick(Column: TColumn);
begin
 if tblIndustry.Active then
  if (Pos(Column.FieldName, tblIndustry.Sort) > 0)and(Pos('ASC', tblIn-
dustry.Sort) > 0)
   then tblIndustry.Sort := Column.FieldName + ' DESC'
   else tblIndustry.Sort := Column.FieldName + ' ASC';
end;
procedure TfmMain.ADOConnAfterConnect(Sender: TObject);
var i: Integer;
begin
  for i := 0 to adoConn.DataSetCount - 1 do
   ADOConn.DataSets[i].Open;
end;
procedure TfmMain.ADOConnBeforeDisconnect(Sender: TObject);
var i: Integer;
begin
  for i := 0 to adoConn.DataSetCount - 1 do
   ADOConn.DataSets[i].Close;
end;
end.
```

Соединение с источником данных

Для связывания приложения с источником данных используем компонент `TADOConnection` и настроим соединения, щелкнув на кнопке свойства `ConnectionString` в Инспекторе объектов.

Перейдя в редактор **Data Link Properties**, выберем провайдер Microsoft OLE DB Provider for OLE DB Drivers (см. рис. 23.3). Как правило, он имеется в операционной системе, если вы не предпринимали специальных усилий по его удалению.

Далее, на странице **Connection** (см. рис. 23.4) выберем переключатель **Use data source name** и в списке — файлы dBase. Для создания соединения с провайдером ODBC этого вполне достаточно.

Прокомментируем другие свойства компонента соединения ADO.

Свойство `LoginPrompt` должно иметь значение `False`, чтобы запретить показ диалога авторизации пользователя, ненужный для фалов dBase.

Свойство `DefaultDatabase` пока останется пустым. Мы применим его для указания пути к файлам базы данных с помощью элементов пользовательского интерфейса приложения.

Свойство `CursorLocation` имеет значение `clUseClient`, чтобы обеспечить использование курсоров наборов данных на стороне клиента.

Свойство `ConnectOptions` имеет значение по умолчанию `coConnectUnspecified`, что означает, что все команды будут выполняться синхронно (соединение будет ожидать ответ на каждую команду).

Для свойства `Mode` установим значение `cmShareDenyNone`, что запрещает другим соединениям устанавливать любые ограничения, ведь в данном случае мы не планируем многопользовательскую работу с источником данных.

Для открытия соединения после запуска приложения необходимо задать путь и хранилищу данных. Для этого предназначены кнопка и однострочный редактор в панели управления. После выбора пути его значение заносится в переменную `DefDBStr` и в текст редактора `edDefDB`. Переменная используется для установления соединения. Для включения соединения необходимо нажать кнопку `tbConnect`. Ее метод-обработчик проверяет состояние переменной `DefDBStr` и заполняет свойство `DefaultDatabase` компонента соединения.

Примечание

Так как во время настройки соединения мы не задавали путь к хранилищу данных, то свойство `DefaultDatabase` сработает. Иначе его значение будет перекрыто настройками свойства `ConnectionString`.

Открытие наборов данных ADO в приложении выполняется в методе-обработчике `ADOConnAfterConnect`, который вызывается после полного открытия соединения. Аналогичным образом наборы данных закрываются перед закрытием соединения в методе-обработчике `ADOConnBeforeDisconnect`.

Текущее значение пути к хранилищу данных сохраняется в файле DemoADO.ini и загружается при открытии приложения.

Реализация групповых операций

Компонент `tblIndustry` предназначен для выполнения групповых операций. Поэтому его свойство `LoclType` имеет значение `ltBatchOptimistic`. Для свойства `CursorLocation` установлено значение `clUseClient`, чтобы обеспечить использование набора данных на стороне клиента. Тип курсора (свойство `CursorType`) должен быть `ctStatic`.

Сохранение изменений в хранилище данных обеспечивает метод `UpdateBatch` в методе-обработчике нажатия кнопки `tbSave`.

Реализация фильтрации

Для фильтрации записей в наборе данных `tblIndustry` служит метод `FilterOnBookmark`. Пользователь должен выбрать интересующие его записи в компоненте `dbgIndustry` (он работает в режиме `dgMultiSelect`). Затем, при нажатии кнопки `tbFilter` созданные в свойстве `SelectedRows` компонента `dbgIndustry` закладки передаются в массив `Bookmarks` типа `TVarRec`, который затем передается в качестве параметра метода `FilterOnBookmark` для фильтрации.

Массив `Bookmarks` служит здесь лишь промежуточным звеном для приведения типа закладок компонента `dbgIndustry` к параметру метода `FilterOnBookmark`.

Реализация сортировки

Сортировка создана также для набора данных `tblIndustry`. При щелчке на заголовке колонки компонента dbgIndustry вызывается метод-обработчик `dbgIndustryTitleClick`. В нем в зависимости от текущего состояния свойства сортировки `tblIndustry.Sort` (какое поле сортируется и в каком порядке) задается новое значение свойства `Sort`.

Резюме

Технология dbGo на основе ADO обеспечивает универсальный способ доступа к гетерогенным источникам данных. Благодаря тому, что функции

ADO реализованы на основе интерфейсов OLE DB и COM, приложению для доступа к данным не требуется дополнительных библиотек, кроме инсталлированного ADO.

Компонент `TADOConnection` обеспечивает соединение с источниками данных через провайдеры OLE DB. Компоненты `TADODataSet`, `TADOTable`, `TADOQuery` и `TADOStoredProc` обеспечивают использование наборов записей в приложении. Свойства и методы компонентов позволяют создавать полнофункциональные приложения.

Компонент `TADOCommand` инкапсулирует текстовую команду ADO.

В дополнение к стандартным возможностям работы с данными, из компонентов можно напрямую обращаться к необходимым объектам и интерфейсам ADO.

ЧАСТЬ VI

Приложения COM и COM+

Глава 24

Механизмы COM

В основе многих технологий, рассматриваемых в этой книге, лежит технология COM (Component Object Model) и технологии, созданные на ее базе. Поэтому мы начнем именно с рассмотрения главных возможностей COM.

Во все времена одной из актуальнейших задач, стоящих перед разработчиками программного обеспечения, была реализация взаимодействия между отдельными программами. Для ее решения создан целый арсенал различных способов и приемов.

На заре существования Windows были внедрены разделяемые файлы, буфер обмена и технология динамического обмена данными (Dynamic Data Exchange, DDE).

Для обеспечения обмена данными и предоставления служб был разработан первый вариант технологии связывания и внедрения объектов (Object Linking and Embedding, OLE 1). Она предназначалась для создания составных документов — того, к чему мы все уже давно привыкли. Эта технология была во многом несовершенной, и на смену ей пришла технология OLE 2. Она позволяет решить более общую проблему — как научить разные программы предоставлять друг другу собственные функции (службы) и как правильно использовать эти функции.

Для решения этой проблемы помимо OLE был разработан целый ряд технологий. В основе этих разработок лежит базовая технология Многокомпонентная модель объектов (Component Object Model, COM). Она описывает способ взаимодействия программ любого типа. Одна часть программного обеспечения предоставляет собственные службы, а другая получает к ним доступ. При этом совершенно не важно, где расположены эти части — в одном процессе или в разных, на одном компьютере или на разных.

Для созданных на основе спецификации COM приложений также не важно, какой язык программирования был выбран при их разработке. Если стандарт COM соблюдается, взаимодействие осуществляется без помех.

Дополнительные возможности разработчикам распределенных приложений на основе COM дает модификация базовой технологии — распределенная модель COM (Distributed COM, DCOM).

В настоящее время COM применяется в самых различных областях разработки программного обеспечения. На основе COM разработаны технологии Автоматизации (Automation), ActiveX, ActiveForm, COM+. Без объектов COM не смогут функционировать распределенные приложения. И при этом не важно, работают ли они в локальной сети или используют Internet.

Delphi предоставляет разработчику набор инструментов для создания полноценных приложений COM.

Далее в этой главе рассматриваются основные части спецификации COM и методика создания объектов и интерфейсов COM в Delphi. Значительное внимание уделяется Редактору библиотеки типов — основному инструменту, который облегчает работу с объектами COM в проекте.

В этой главе рассматриваются следующие вопросы.

- Объекты и интерфейсы.
- Интерфейс `IUnknown`.
- Библиотеки типов.
- Фабрики классов.
- Виды серверов.

Базовые понятия

В технологии COM приложение предоставляет для использования свои службы, применяя для этого *объекты COM*. Одно приложение содержит как минимум один объект. Каждый объект имеет один или несколько *интерфейсов*. Каждый интерфейс объединяет *методы* объекта, которые обеспечивают доступ к *свойствам* (данным) и выполнение операций. Обычно в интерфейсе объединяются все методы, выполняющие операции одного типа или работающие с однородными свойствами.

Клиент получает доступ к службам объекта только через интерфейс и его методы. Этот механизм является ключевым. Клиенту достаточно знать несколько базовых интерфейсов, чтобы получить исчерпывающую информацию о составе свойств и методов объекта. Поэтому любой клиент может работать с любым объектом, независимо от их среды разработки. Согласно спецификации COM, уже созданный интерфейс не может быть изменен ни при каких обстоятельствах. Это гарантирует постоянную работоспособность приложений на основе COM, невзирая на любые модернизации.

Объект всегда работает в составе *сервера COM*. Сервер может быть динамической библиотекой или исполняемым файлом. Объект может иметь собственные свойства и методы или использовать данные и службы сервера.

Для доступа к методам объекта клиент должен получить указатель на соответствующий интерфейс. Для каждого интерфейса существует собственный

указатель. После этого клиенту доступны службы объекта через вызов его методов. Доступ к свойствам объектов также осуществляется только через его методы.

Предположим, что объект COM встроен в электронную таблицу и обеспечивает доступ к математическим операциям. Будет логично разделить математические функции на группы по типам и создать для каждой группы собственный интерфейс. Например, можно выделить линейные, тригонометрические, агрегатные функции и т. д. На рис. 24.1 объект расположен внутри сервера (электронной таблицы). Интерфейсы обозначены маленькими кружками, связанными с объектом. Пусть интерфейс линейных функций называется `ILinear`, а агрегатных — `IAggregate`.

Примечание

Согласно принятым в Delphi правилам именования, имена интерфейсов начинаются с заглавной буквы I. Напомним также, что имена классов начинаются с заглавной буквы T.

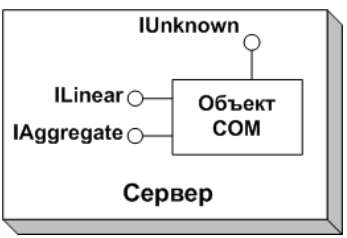


Рис. 24.1. Сервер, объект и его интерфейсы

Примечание

Согласно правилам обозначения объектов COM, базовый интерфейс `IUnknown` (см. далее), который имеется у любого объекта, обозначается в виде кружка, примыкающего к верхней стороне прямоугольника объекта. Остальные интерфейсы обозначаются справа или слева.

На рис. 24.2 представлена схема взаимодействия клиента с объектом COM.

Предположим, что среди методов интерфейса `IAggregate` имеется метод вычисления среднего. Чтобы получить доступ к агрегатной функции расчета среднего, клиент должен получить указатель на интерфейс `IAggregate`, а затем обратиться к этой функции.

Взаимодействие между клиентом и объектом обеспечивается базовыми механизмами COM. При этом от клиента скрыто, где именно расположен объект: в адресном пространстве того же процесса, в другом процессе или на другом компьютере. Поэтому с точки зрения разработчика клиентского ПО

вызов функций электронной таблицы выглядит как обычное обращение к методу объекта. Механизм обеспечения взаимодействия между удаленными элементами COM называется *маршалингом* (marshalling).

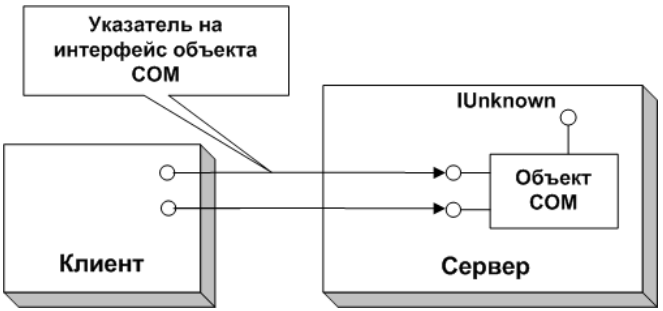


Рис. 24.2. Схема взаимодействия клиента и объекта COM

Возникает закономерный вопрос: как проходит создание и инициализация объекта COM при первом обращении клиента? Сомнительно, чтобы операционная система самостоятельно создавала экземпляры всех зарегистрированных в ней классов в надежде, что один из них понадобится. А ведь для работы объекта требуются еще и серверы. Представьте, что каждый раз при загрузке Windows настойчиво запускает Word, Excel, Internet Explorer и т. д.

Любой объект COM является обычным экземпляром некоторого класса, описывающего его свойства и методы. Информация обо всех зарегистрированных и доступных в данной операционной системе классах COM собрана в специальной *библиотеке COM*, которая служит для запуска экземпляра класса — объекта.

Сначала клиент обращается к библиотеке COM, передавая ей имя требуемого класса и необходимого в первую очередь интерфейса. Библиотека находит нужный класс и сначала запускает сервер, который затем создает объект (экземпляр класса). После этого библиотека возвращает клиенту указатели на объект и интерфейс. В последующей работе клиент может обращаться непосредственно к объекту и его интерфейсам.

После создания наступает очередь инициализации: объект должен загрузить необходимые данные, считать настройки из системного реестра и т. д. За это отвечают специальные объекты COM, которые называются *моникерами* (monikers). Они работают скрытно от клиента. Обычно моникер создается вместе с классом.

Довольно реальной представляется ситуация, когда одновременно несколько клиентов обращаются к одному объекту. При соответствующих настройках для каждого клиента создается отдельный экземпляр класса. За выполнение

этой операции отвечает специальный объект COM, который называется *фабрикой класса*.

Наконец, остался еще не рассмотренным последний вопрос: как клиент может получить информацию об объекте. Например, разработчик клиентского ПО знает, что электронная таблица создана в соответствии со спецификацией COM, но не имеет понятия об объектах COM, которые предоставляют клиентам ее службы. Для разрешения подобных ситуаций разработчик объекта COM может распространять вместе с объектом информацию о типе. Она включает сведения об интерфейсах, их свойствах и методах, параметрах методов.

Эта информация содержится в *библиотеке типов*, которая создается при помощи специального *языка описания интерфейса* (Interface Definition Language, IDL).

Объект

Центральным элементом COM является объект. Приложения, поддерживающие COM, имеют в своем составе один или несколько объектов COM. Каждый объект представляет собой экземпляр соответствующего класса и содержит один или несколько интерфейсов. Что же такое объект COM?

Не вдаваясь пока в подробности реализации объектов COM в Delphi, можно сказать, что объект COM несколько отличается от обычного объекта.

Любой объект является экземпляром некоторого класса, т. е. представляет собой переменную объектного типа. Поэтому объект обладает набором свойств и методов, которые работают с этими свойствами. К объектам применимы три основные характеристики: инкапсуляция, наследование и полиморфизм. Объекты COM всем этим требованиям удовлетворяют (существуют особенности наследования).

Для объектов вообще понятие интерфейса объекта, как он был определен ранее, не применимо. В первом приближении можно считать, что все методы объекта составляют его единственный интерфейс, а указателем интерфейса является указатель на объект.

Объект COM может иметь любое число интерфейсов (если это число больше нуля), причем каждый интерфейс обладает собственным указателем. Это первое отличие объектов COM от обычных.

Примечание

Некоторые языки программирования, например, Java, позволяют объекту иметь несколько интерфейсов.

У объектов COM имеется особенность еще в одном объектном механизме — наследовании. Вообще различают два способа наследования. Наследование

реализации подразумевает передачу родителем потомку всего программного кода. Наследование интерфейса означает передачу только объявления методов, их программный код потомок должен предоставить самостоятельно.

Объекты COM поддерживают только наследование интерфейса, избегая тем самым возможного нарушения инкапсуляции родителя. Тем не менее просто так выбросить наследование реализации нельзя. Вместо нее объекты COM имеют механизм *включения*, т. е. при необходимости потомок вызывает нужный метод родителя. Также применяется механизм *агрегирования*, когда один или несколько интерфейсов одного объекта на время включаются в другой объект путем передачи указателей.

Таким образом, объект COM с точки зрения ООП несомненно является объектом. Однако как ключевой элемент технологии COM он обладает рядом особенностей реализации базовых механизмов.

Интерфейс

Если объект COM является ключевым элементом реализации COM, то интерфейсы — центральное звено идеологии COM. Как двум принципиально разным объектам обеспечить взаимодействие друг с другом? Ответ прост: им необходимо заранее договориться о том, как они будут общаться. (Авторы намеренно не используют слово "язык", т. к. оно может вызвать нежелательные ассоциации с языком программирования, а этот фактор не имеет во взаимодействии элементов COM никакого значения.)

Интерфейс как раз является тем средством, которое позволяет клиенту правильно обратиться к объекту COM, а объекту ответить так, чтобы клиент его понял.

Рассмотрим небольшой пример. На улице случайно встретились два человека: местный житель (объект COM) и заблудившийся иностранец (клиент). Предусмотрительный иностранец захватил с собой словарь (библиотека типов или интерфейс `IUnknown`). Иностранцу нужны знания местного жителя о городе. Он достал ручку и бумагу и, заглянув в словарь, составил фразу и старательно перерисовал незнакомые слова на бумагу. Местный житель оказался не промах. Он прочитал фразу, отобрал у иностранца словарь, составил по нему собственную фразу и тоже написал ее на бумаге.

И все закончилось хорошо: довольный клиент (иностранец) получил от объекта COM (местного жителя) результат работы службы (информацию о дороге), а местный житель ушел вместе со словарем.

Как вы уже догадались, в этом примере интерфейсом является бумага и ручка. Иностранец не знает чужого языка, зато знает, *как* правильно спросить, чтобы ему ответили.

Для идентификации каждый интерфейс имеет два атрибута. Во-первых, это его имя, составленное из символов в соответствии с правилами выбранного языка программирования. Каждое имя должно начинаться с символа "I". Это имя используется в программном коде. Во-вторых, это *глобальный уникальный идентификатор* (Globally Unique IDentifier, GUID), который представляет собой гарантированно уникальное сочетание символов, практически не повторяемое ни на одном компьютере в мире. Для интерфейсов такой идентификатор носит название IID (Interface Identifier).

В общем случае клиент может не знать, какие интерфейсы имеются у объекта. Для получения их перечня необходим базовый интерфейс `IUnknown` (см. далее), который есть у любого объекта COM.

Затем клиенту необходимо знать, какие методы имеет выбранный им интерфейс. Для этого разработчик должен распространять описание методов интерфейсов вместе с объектом. Эту задачу помогает решать язык IDL (он также имеется в библиотеках типов). Его синтаксис очень похож на C++.

Теперь осталось сделать самое важное — правильно вызвать сам метод. Для этого в COM описана реализация интерфейса на основе стандартного двоичного формата. Это обеспечивает независимость от языка программирования.

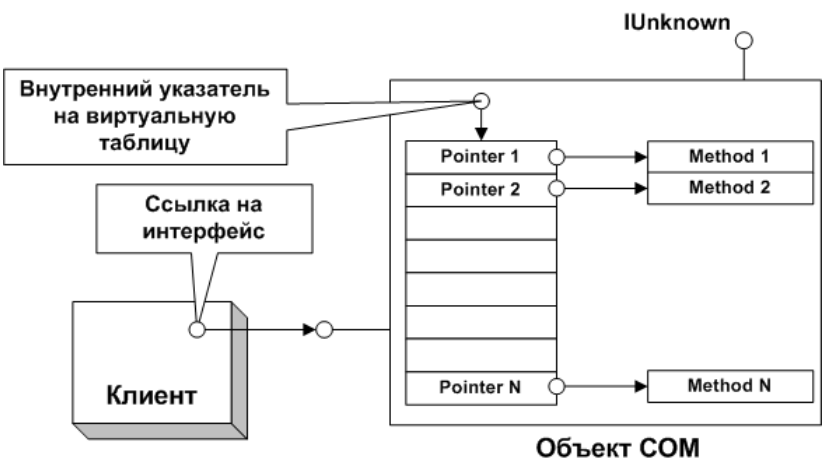


Рис. 24.3. Формат интерфейса COM

Доступный клиенту указатель интерфейса ссылается на внутренний указатель объекта и, через него, на специальную виртуальную таблицу (рис. 24.3). Эта таблица содержит указатели на все методы интерфейса. (Не правда ли, очень похоже на таблицу виртуальных методов объекта в ООП.)

Примечание

Первые три строки таблицы интерфейса всегда заняты под методы интерфейса `IUnknown`, так как любой интерфейс COM является т. к. наследником этого интерфейса.

В результате, вызов метода клиентом проходит по цепочке указателей и получает указатель на конкретный метод, а затем исполняется соответствующий программный код.

Интерфейс *IUnknown*

Каждый объект COM обязательно имеет интерфейс `IUnknown`. Этот интерфейс имеет всего три метода, но они играют ключевую роль в функционировании объекта.

Метод `QueryInterface` возвращает указатель на интерфейс объекта, идентификатор IID которого передается в параметре метода. Если такого интерфейса объект не имеет, метод возвращает `NULL`.

Обычно при первом обращении к объекту клиент получает указатель на интерфейс. Так как любой интерфейс является потомком `IUnknown`, то любой интерфейс имеет и метод `QueryInterface`. Поэтому в общем случае не важно, какой именно интерфейс есть у клиента. При помощи метода `QueryInterface` он может получить доступ к любому интерфейсу объекта.

Интерфейс `IUnknown` обеспечивает работу еще одного важного механизма объекта COM — механизма учета ссылок. Объект должен существовать до тех пор, пока его использует хотя бы один клиент. При этом клиент не может самостоятельно уничтожить объект, ведь с ним могут работать и другие клиенты.

Поэтому при передаче наружу очередного указателя на интерфейс объект увеличивает специальный счетчик ссылок на единицу. Если один клиент передает другому указатель на интерфейс этого объекта, то клиент, получающий указатель, обязан еще раз инкрементировать счетчик ссылок. Для этого применяется метод `AddRef` интерфейса `IUnknown`.

При завершении работы с интерфейсом клиент обязан вызвать метод `Release` интерфейса `IUnknown`. Этот метод уменьшает счетчик ссылок на единицу. После обнуления счетчика объект уничтожает себя.

Сервер

Сервер COM представляет собой исполняемый файл (приложение или динамическую библиотеку), который может содержать один или несколько объектов одного или разных классов.

Различают три типа серверов.

- Внутренний сервер (in-process server) — реализуется динамическими библиотеками, которые подключаются к приложению-клиенту и работают в одном с ним адресном пространстве.
- Локальный сервер (local server) — создается отдельным процессом, который работает на одном компьютере с клиентом.
- Удаленный сервер (remote server) — создается процессом, который работает на другом компьютере по отношению к клиенту.

Рассмотрим локальный сервер. Получаемый клиентом указатель интерфейса в этом случае ссылается на специальный *proxy-объект* COM (назовем его *заместителем*), который функционирует внутри клиентского процесса. Заместитель предоставляет клиенту те же интерфейсы, что и вызываемый объект COM на локальном сервере. Получив вызов от клиента, заместитель упаковывает его параметры и при помощи служб операционной системы передает вызов в процесс сервера. В локальном сервере вызов передается еще одному специализированному объекту — *заглушке* (stub), который распаковывает вызов и передает его требуемому объекту COM. Результат вызова возвращается клиенту в обратном порядке.

Рассмотрим удаленный сервер. Он функционирует так же, как и локальный. Однако передача вызовов между двумя компьютерами осуществляется средствами DCOM — с помощью механизма *вызова удаленных процедур* (Remote Procedure Call, RPC).

Для обеспечения работы локальных и удаленных серверов предназначен механизм маршалинга и демаршалинга. *Маршалинг* реализует единый в рамках COM формат упаковки параметров запроса, *демаршалинг* отвечает за распаковку. В описанных реализациях серверов за выполнение этих операций отвечают заместитель и заглушка. Эти типы объектов создаются совместно с основным объектом COM. Для этого применяется IDL.

Библиотека COM

Для обеспечения выполнения базовых функций и интерфейсов в операционной системе существует специальная библиотека COM (конкретная реализация может быть различной). Доступ к возможностям библиотеки осуществляется стандартным способом — через вызов функций. Согласно спецификации, имена всех библиотечных функций начинаются с приставки `Co`.

При установке поддерживающего COM приложения в системный реестр записывается информация обо всех реализуемых им объектах COM.

- Идентификатор класса (Class Identifier, CLSID), который однозначно определяет класс объекта.

- Тип сервера объекта (внутренний, локальный или удаленный).
- Для локальных и внутренних серверов сохраняется полное имя динамической библиотеки или исполняемого файла.
- Для удаленных серверов записывается полный сетевой адрес.

Предположим, что клиент пытается вызвать некоторый объект COM, который до этого момента не использовался. Следовательно, клиент не имеет указателя на нужный объект и интерфейс. В этом случае он обращается к библиотеке COM и вызывает специальный метод `CoCreateInstance`, передавая ему в качестве параметра CLSID нужного класса, IID интерфейса и требуемый тип сервера.

Библиотека при помощи диспетчера управления службами (Service Control Manager, SCM) обращается к системному реестру, по идентификатору класса находит информацию о сервере и запускает его. Сервер создает экземпляр класса — объект, и возвращает библиотеке указатель на запрошенный интерфейс.

Библиотека COM передает указатель клиенту, который впоследствии может обращаться непосредственно к объекту. Схема создания первого экземпляра объекта с помощью библиотеки COM и системного реестра приведена на рис. 24.4.

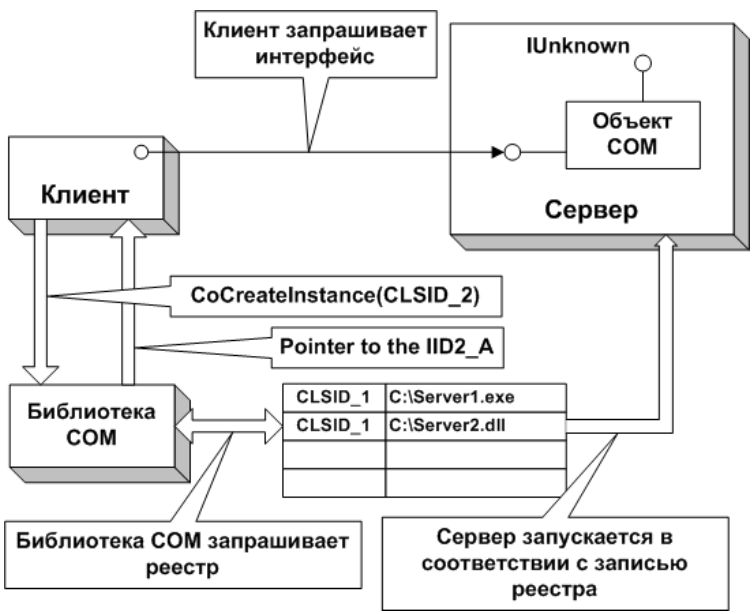


Рис. 24.4. Создание первого экземпляра объекта с помощью библиотеки COM и системного реестра

Примечание

На рис. 24.4 клиент использует метод `CoCreateInstance`, вызывая объект COM с `GUID CLSID_2` и запрашивая интерфейс `IID2_A`. По глобальному идентификатору `CLSID_2` находится соответствующая запись в системном реестре и клиенту возвращается указатель на искомый интерфейс.

Неявную инициализацию созданного объекта (установки значений свойств) может реализовать особый объект — моникер. Также клиент может инициализировать объект самостоятельно, применив специальные интерфейсы (`IPersistFile`, `IPersistStorage`, `IPersistStream`).

Фабрика класса

Для запуска экземпляра класса есть специальный объект — фабрика класса. С его помощью можно создать как один объект, так и несколько его экземпляров. Для каждого класса должна существовать собственная фабрика класса.

Согласно спецификации COM фабрика класса также является объектом COM. И объект COM имеет право называться фабрикой класса, если он поддерживает интерфейс `IClassFactory`. В нем реализованы всего два метода.

- `CoCreateInstance` — создает новый экземпляр класса. Все необходимые параметры, кроме `IID`, метод получает от фабрики класса. В этом его отличие от одноименной общей функции библиотеки.
- `LockServer` — оставляет сервер функционировать после создания объекта.

Примечание

На самом деле общий метод `CoCreateInstance` при помощи переданного ему `CLSID` осуществляет вызов соответствующей фабрики класса и метода `CoCreateInstance` интерфейса `IClassFactory`.

Для вызова фабрики класса существует специальная функция `CoGetClassObject`:

```
function CoGetClassObject(const clsid: TCLSID; dwClsContext: Longint;
pvReserved: Pointer; const iid: TIID; var pv): HResult; stdcall;
```

В качестве параметра ей передается `CLSID` нужного класса и `IID` интерфейса (`IClassFactory`). Функция ищет требуемую фабрику и возвращает указатель на интерфейс. С его помощью через метод `CoCreateInstance` клиент заставляет фабрику класса создать объект.

Библиотека типов

Чтобы документировать интерфейсы объекта для пользователей, разработчик создает (на языке IDL) информацию о типах объекта, которая объединяется в специальной библиотеке типов. Она может описывать свойства и методы (а также их параметры) интерфейсов и содержать сведения о необходимых заглушках и заместителях. Информация об отдельном интерфейсе оформляется в виде отдельного объекта внутри библиотеки.

Для создания библиотеки типов, описанной при помощи операторов IDL, разработаны специальные компиляторы. Доступ к библиотеке осуществляется по `CLSID` класса объекта. Кроме того, библиотека имеет собственный GUID, который сохраняется в системном реестре при регистрации объекта.

Каждая библиотека типов имеет интерфейс `ITypeLib`, который дает возможность работать с ней, как с единым объектом. Для доступа к информации об отдельном интерфейсе применяется интерфейс `ITypeInfo`.

Доступ к библиотеке по GUID обеспечивает функция `LoadRegTypeLib`. Если клиенту известно имя файла библиотеки, то можно воспользоваться функцией `LoadTypeLib`.

Объекты COM в Delphi

Теперь рассмотрим механизм создания объектов COM в Delphi. Как уже говорилось, объект COM должен обеспечивать возможность создания произвольного числа интерфейсов, где под интерфейсом понимается некоторое объединение методов, доступ к которым осуществляется через указатель на интерфейс.

Реализовать такое требование напрямую в рамках стандартных подходов ООП довольно затруднительно. И разработчики Delphi нашли следующий выход.

Сам объект COM описывается обычным классом `TComObject`, который порожден непосредственно от `TObject`. Все свойства и методы, реализующие предназначение объекта, объявляются и описываются в его объявлении. Поэтому сам класс нового объекта COM принципиально ничем не отличается от любого другого.

При создании объекта COM с ним связывается еще один вспомогательный класс, который описывает все интерфейсы. Этот класс носит общее название `CoClass`, а при создании реального объекта к его имени добавляется приставка `Co`. `CoClass` объединяет всю информацию о типах, которая представлена в библиотеке типов. Объявление и описание `CoClass` содержится в библиотеке типов.

Итак, стандартное объявление класса на Object Pascal обеспечивает создание кода объекта — именно оно компилируется в двоичный код. `CoClass` является надстройкой или оболочкой этого кода, он обеспечивает представление экземпляра объекта в соответствии со спецификацией COM и гарантирует, что обращение клиента к объекту будет обработано корректно.

Класс `TComObject` в совокупности с создаваемым для каждого объекта экземпляром `CoClass` обладает всеми рассмотренными выше признаками объекта COM. Он может поддерживать произвольное число интерфейсов и в том числе базовый интерфейс `IUnknown`.

Для обеспечения работы объекта COM с библиотекой типов от базового класса `TComObject` порожден новый класс `TTypedComObject`. Дополнительно он имеет еще один интерфейс — `IProvideClassInfo`. Если этот интерфейс имеет доступ к библиотеке типов, то для получения полной информации об объекте достаточно знать его идентификатор класса. Этот класс нужен для создания объектов на основе библиотеки типов.

Класс *TComObject*

Класс `TComObject` обеспечивает выполнение базовых функций объекта, которые и делают его объектом COM. Его свойства и методы инкапсулируют функциональность интерфейса `IUnknown`. Он обеспечивает также хранение идентификатора класса `CLSID`.

Как и любой другой объект, COM объект создается конструктором

```
constructor Create;
```

который создает самостоятельный экземпляр класса.

В то же время конструктор

```
constructor CreateAggregated(const Controller: IUnknown);
```

создает новый объект как часть агрегата. В данном случае под агрегатом понимается совокупность объектов, предоставляющих собственные интерфейсы, за исключением одного общего. Этим общим интерфейсом является интерфейс `IUnknown`, который, как известно, имплементирует счетчик обращений и управляет всеми объектами агрегата. Объект, предоставляющий общий управляющий интерфейс, называется контроллером.

В конструкторе `CreateAggregated` параметр `Controller` определяет управляющий интерфейс и после вызова конструктора он передается в свойство

```
property Controller: IUnknown;
```

Для реального создания объекта и инициализации его параметров используется конструктор

```
constructor CreateFromFactory(Factory: TComObjectFactory; const
Controller: IUnknown);
```

Параметр `Factory` позволяет задать свойство

```
property Factory: TComObjectFactory;
```

определяющее фабрику класса для объекта. При создании объекта метод

```
procedure Initialize; virtual;
```

проводит инициализацию. Для класса `TComObject` этот метод пуст, однако он может быть перекрыт в потомках.

Оба других конструктора всего лишь вызывают конструктор `CreateFromFactory`, каждый по-своему инициализируя переменную `Controller`, определяющую принадлежность к агрегату:

```
constructor TComObject.Create;
begin
  FNonCountedObject := True;
  CreateFromFactory(ComClassManager.GetFactoryFromClass(ClassType), nil);
end;

constructor TComObject.CreateAggregated(const Controller: IUnknown);
begin
  FNonCountedObject := True;
  CreateFromFactory(ComClassManager.GetFactoryFromClass(ClassType),
  Controller);
end;
```

После создания объекта для чтения доступно свойство

```
property RefCount: Integer;
```

которое возвращает число открытых ссылок на объект. Значение этого свойства определяется исходя из вызовов интерфейса `IUnknown`.

Сами же методы интерфейса `IUnknown` объекта доступны через вызовы одноименных методов класса `TComObject`.

Метод

```
function ObjAddRef: Integer; virtual; stdcall;
```

увеличивает счетчик ссылок на единицу. Соответственно увеличивается и свойство `RefCount`.

Метод

```
function ObjQueryInterface(const IID: TGUID; out Obj): HResult; virtual;
stdcall;
```

позволяет определить, используется ли объектом интерфейс, заданный параметром `IID`.

И метод

```
function ObjRelease: Integer; virtual; stdcall;
```

уменьшает счетчик ссылок на единицу. Соответственно уменьшается и свойство `RefCount`.

Класс *TTypedComObject*

Этот класс является непосредственным потомком класса `TComObject` и применяется для создания объектов COM на основе библиотеки типов. Класс обладает дополнительным интерфейсом `IProvideClassInfo`, который обеспечивает работу единственного нового метода:

```
function GetClassInfo(out TypeInfo: ITypeInfo): HResult; stdcall;
```

Эта функция возвращает указатель на `CoClass` конкретного класса, открывая тем самым доступ ко всей информации о типах.

Интерфейс *IUnknown* в Delphi

Класс `TComObject` имеет методы, которые соответствуют методам инкапсулированного интерфейса `IUnknown`.

В качестве общего предка всех интерфейсов в Delphi выступает интерфейс

```
IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

который, как видно из объявления, содержит все три метода, обеспечивающие работу интерфейса `IUnknown`. Поэтому интерфейс `IUnknown` объявлен так:

```
IUnknown = IInterface;
```

Метод

```
function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
```

возвращает указатель на интерфейс `IID`, если он доступен.

Метод

```
function _AddRef: Integer; stdcall;
```

увеличивает счетчик ссылок на единицу.

Метод

```
function _Release: Integer; stdcall;
```

ументшает счетчик ссылок на единицу.

Согласно спецификации COM, в Delphi любые интерфейсы являются потомками интерфейса `IUnknown`.

Тип глобального идентификатора

Для представления глобального уникального идентификатора `GUID` в Delphi в пространстве имен `System` определен специальный тип:

```
TGUID = record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

Сгенерировать новый идентификатор можно с помощью функции `CoCreateGUID` из Win API.

Для преобразования идентификатора в строку существует функция

```
function GUIDToString(const ClassID: TGUID): string;
```

Для обратной операции применяется функция

```
function StringToGUID(const S: string): TGUID;
```

Класс *TInterfacedObject*

Специально для работы с объектами COM, инкапсулирующими интерфейсы, в иерархии базовых классов Delphi предусмотрен класс `TInterfacedObject`, который обеспечивает создание класса-потомка, наследующего свойства и методы не только от родительского класса, но и методы от родительского интерфейса:

```
TInterfacedObject = class(TObject, IInterface)
```

В объявлении всех классов, произошедших от класса `TInterfacedObject`, должны указываться не только класс-предок, но и интерфейс-предок. В результате новый класс наследует также и методы родительского интерфейса.

Создание экземпляра такого класса выполняется без привлечения фабрики класса, а обычным конструктором:

```
type
  TSomeInterfacedObject = class(TInterfacedObject, IUnknown)
```

```
  end;
...
var SomeInterfacedObject: TsomeInterfacedObject;
...
SomeInterfacedObject := TSomeInterfacedObject.Create;
```

Так как класс `TInterfacedObject` наследует от интерфейса `IInterface` (он же `IUnknown`), он содержит свойство счетчика ссылок

```
property RefCount: Integer;
```

и базовых три метода:

```
function _AddRef: Integer; stdcall;
function _Release: Integer; stdcall;
function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
```

Как известно, выделение памяти под новый объект осуществляется методом класса `NewInstance`. Для класса `TInterfacedObject` метод

```
class function NewInstance: TObject; override;
```

помимо основной обязанности дополнительно инициализирует счетчик ссылок.

Фабрика класса в Delphi

При реализации объектов COM в Delphi разработчик не должен заботиться о создании фабрики класса для каждого объекта. Как будет показано далее (см. листинг 24.1), совместно с объектом создается не только `CoClass`, но и фабрика класса — специализированный объект COM, который обладает интерфейсом `IClassFactory` и используется для создания объектов COM.

Внимательный читатель уже обратил внимание, что при создании экземпляра базового класса `TComObject` в любом случае вызывается конструктор `CreateFromFactory`, который отвечает за вызов соответствующей фабрики класса. Объект COM связывается со своей фабрикой класса при помощи свойства `Factory`, которое имеет тип `TComObjectFactory`. Класс `TComObjectFactory` — основа для применения фабрик классов в приложениях COM в Delphi.

В случаях, когда в рамках одного сервера функционирует несколько фабрик классов, необходим специальный диспетчер классов COM. Он создается на базе класса `TComClassManager` и служит в основном для внутренних операций, когда серверу требуется выполнить определенные действия со всеми фабриками классов (например, корректно уничтожить их).

Для создания фабрики класса, объявленного с помощью класса `TTypedComObject`, имеется класс `TTypedComObjectFactory`.

Класс *TComObjectFactory*

Этот класс инкапсулирует функции универсальной фабрики класса для объектов COM, создаваемых как экземпляры класса `TComObject`. Он обеспечивает функционирование интерфейсов `IUnknown`, `IClassFactory` и `IClassFactory2`. Определяющим в работе фабрики класса является интерфейс `IClassFactory`.

Фабрика класса создается обычно при функционирующем сервере создаваемого ею объекта. Конструктор фабрики описывается в секции инициализации модуля, включающего соответствующий сервер (см. листинг 24.1).

Фабрика класса создается конструктором

```
constructor Create(ComServer: TComServerObject; ComClass: TComClass;
const ClassID: TGUID; const ClassName, Description: string; Instancing:
TClassInstancing; ThreadingModel: TThreadingModel = tmSingle);
```

Здесь:

- `ComServer` — определяет сервер, в котором будет функционировать объект;
- `ComClass` — определяет тип класса, который используется для идентификации фабрики класса во время работы процедуры `GetFactoryFromClass` класса `TComClassManager`;
- `ClassID` — задает идентификатор класса, который создается при помощи этой фабрики класса;
- `Instancing` — определяет способ создания объекта;
- `ThreadingModel` — определяет способ взаимодействия объекта и клиента.

После создания фабрики класса основные параметры создаваемого ею объекта COM доступны через свойства.

Свойство

```
property ClassID: TGUID;
```

возвращает GUID класса объекта.

Свойство

```
property ClassName: string;
```

содержит имя класса объекта.

Свойство

```
property ComClass: TClass;
```

определяет тип класса, который служит для идентификации фабрики класса.

Способ создания объекта определяется свойством

```
type TClassInstancing = (ciInternal, ciSingleInstance, ciMultiInstance);
property Instancing: TClassInstancing;
```

А свойство

```
property ThreadingModel: TThreadingModel;
```

задает способ взаимодействия объекта и клиента.

Ключевым методом класса является функция

```
function CreateComObject(const Controller: IUnknown): TComObject;
```

Она выполняет действия функции `CoCreateInstance` интерфейса `IClaccFactory`. Метод вызывает конструктор `CreateFromFactory` связанного с данной фабрикой класса, передавая в него необходимые параметры.

Метод

```
procedure RegisterClassObject;
```

выполняет регистрацию класса создаваемого объекта. Это делается при запуске исполняемого файла, инкапсулирующего объект COM.

Метод

```
procedure UpdateRegistry(Register: Boolean); virtual;
```

регистрирует или разрегистрирует объект COM. Для исполняемых приложений это делается при первом запуске или при запуске с ключом `/regserver`. Метод обеспечивает сохранение в системном реестре основных параметров объекта COM и местоположение исполняемого файла.

При создании объекта COM разработчик может предусмотреть специальный интерфейс, отвечающий за обработку ошибок. Его GUID задается свойством

```
property ErrorIID: TGUID;
```

а свойство

```
property ShowErrors: Boolean;
```

включает или отключает отображение информации об ошибках при создании объекта.

При распространении объекта COM разработчик может включить в исполняемый код лицензионную информацию и сведения о разработчике.

Текстовое описание объекта COM можно задать, заполнив свойство

```
property Description: string;
```

Свойство

```
property ProgID: string;
```

содержит имя разработчика, класса и версию.

Свойство

```
property LicString: WideString;
```

должно содержать лицензионную информацию об объекте.

А если разработчик предусмотрел процедуру регистрации пользователей и выполнения лицензионного соглашения, свойство

```
property SupportsLicensing: Boolean;
```

позволяет задать режим запуска с лицензированием. Для этого свойство должно иметь значение `True`.

Класс *TTypedComObjectFactory*

Этот класс порожден от класса `TComObjectFactory` и используется для создания фабрик классов, классы которых объявлены с помощью класса `TTypedComObject`. То есть этот класс применяется для описания фабрики класса в библиотеке типов. Класс `TTypedComObjectFactory` имеет один дополнительный метод:

```
function GetInterfaceTypeInfo(TypeFlags: Integer): ITypeInfo;
```

Функция возвращает указатель на интерфейс `ITypeInfo`, который содержит информацию об отдельном типе.

Класс *TComClassManager*

Этот класс служит для управления фабриками класса в составе отдельного сервера COM. Он представляет собой класс типизированной переменной `ComClassManager` в составе модуля ComObj.

Получить ссылку на экземпляр класса можно при помощи функции того же модуля

```
function ComClassManager: TComClassManager;
```

Разработчику может быть полезным метод класса:

```
function GetFactoryFromClassID(const ClassID: TGUID): TComObjectFactory;
```

который проводит поиск по идентификатору класса `ClassID` среди работающих фабрик классов сервера.

Метод

```
function GetFactoryFromClass(ComClass: TClass): TComObjectFactory;
```

возвращает ссылку на фабрику класса для класса `ComClass`. В частности, этот метод применяется в конструкторе объектов COM для получения ссылки на фабрику класса.

Сервер COM в Delphi

При создании объекта COM в модуль с его описанием автоматически добавляется модуль `ComServ`. Этот модуль описывает класс `TComServer`, который инкапсулирует свойства сервера COM, в котором работает соответствующий объект. Обращение к свойствам и методам этого класса позволяет получить информацию о работающих в сервере объектах и их состоянии, а также о самом сервере.

При включении модуля `ComServ` в модуль объекта автоматически создается экземпляр класса `TComServer`, указатель на который присваивается переменной

```
var ComServer: TComServer;
```

Через эту переменную можно получать информацию от сервера.

Класс сервера служит для создания экземпляров фабрик классов, т. е. непосредственно участвует в работе механизма взаимодействия объектов COM и клиентов. В модуле `ComServ` объявлены и описаны глобальные функции, которые автоматически экспортируются в каждый внутренний сервер и выполняют базовые операции регистрации, перерегистрации и выгрузки сервера. Вызывать их напрямую нет необходимости.

Класс *TComServer*

Этот класс объединяет свойства и методы, которые позволяют получить информацию о самом сервере и функционировании объектов в сервере COM. Класс является потомком абстрактного класса `TComServerObject`.

Класс предназначен для внутренних и локальных серверов.

Если свойство

```
property IsInprocServer: Boolean;
```

имеет значение `True`, сервер является внутренним. Но более точно определить тип сервера поможет свойство

```
property ServerKey: string;
```

если в нем содержится значение `'InprocServer32'`, то сервер является внутренним и представляет собой DLL. Если вы видите значение `'LocalServer32'`, то это локальный сервер в виде исполняемого файла.

А способ запуска файла сервера задан свойством

```
type TStartMode = (smStandalone, smAutomation, smRegServer,
smUnregServer);
property StartMode: TStartMode;
```

которое может иметь следующие значения:

- `smStandalone` — сервер запускается как отдельное приложение;
- `smAutomation` — сервер запускается как сервер Автоматизации *(см. главу 25)*;
- `smRegServer` — сервер запускается с ключом `/regserver` (или запускается первый раз), для того, чтобы зарегистрироваться в системном реестре;
- `smUnregServer` — сервер запускается с ключом `/unregserver` для разрегистрации.

При запуске сервера вызывается метод

```
procedure Initialize;
```

который проводит регистрацию (при первом запуске или использовании ключа `/regserver`) всех объектов COM, связанных с данным сервером. Например, это могут быть главный и дочерний удаленные модули данных — специализированные объекты COM для технологии DataSnap *(см. главу 27)*.

Имя сервера задано свойством только для чтения

```
property ServerName: string;
```

А задать его можно методом

```
procedure SetServerName(const Name: string);
```

Однако обратите внимание, что если сервер работает с библиотекой типов, то имя сервера определяется ею. Ссылка на библиотеку типов доступна через свойство

```
property TypeLib: ITypeLib;
```

возвращающее интерфейс `ITypeLib` — основной интерфейс библиотеки типов.

Загрузить библиотеку типов можно при помощи метода

```
procedure LoadTypeLib;
```

Полное имя файла сервера содержится в свойстве

```
property ServerFileName: string;
```

а при наличии файла помощи его полное имя задается свойством

```
property HelpFileName: string;
```

Еще одно полезное свойство только для чтения

```
property ObjectCount: Integer;
```

необходимо в процессе работы сервера. Оно возвращает общее число объектов COM, работающих с данным сервером.

Библиотека типов в Delphi

Библиотеки типов обеспечивают хранение информации об объектах, интерфейсах, функциях и т. д. Они применяются в проектах Delphi, основанных на технологиях COM.

Традиционно для создания библиотек типов употребляется язык описания интерфейсов IDL (Interface Description Language). В Delphi в качестве основного варианта применяются синтаксис и операторы языка Delphi. Однако вы всегда можете экспортировать этот код в формат IDL, обеспечив тем самым реализацию собственных объектов в любых приложениях Windows.

Библиотека типов необходима клиентам при обращении к объекту для получения начальной информации о доступных идентификаторах, интерфейсах, методах. В принципе, эти же данные можно получить и программными средствами из системного реестра и базовых интерфейсов COM. Однако не всегда удобно дополнять приложение довольно сложным блоком кода из-за необходимости применить небольшой объект для вспомогательных целей.

Обычно внутри проекта используется вариант библиотеки на языке Delphi, а при распространении объектов библиотека типов экспортируется в формат IDL. Вариант библиотеки типов на языке Delphi сохраняется в файле с расширением pas и окончанием _TLB в названии.

В Delphi код библиотеки типов генерируется автоматически при создании объекта. Для работы с библиотекой предусмотрен специальный Редактор библиотеки типов (рис. 24.5). Все выполняемые в нем операции модифицируют исходный код библиотеки типов и соответствующих объектов и интерфейсов, поэтому разработчику нет необходимости досконально изучать особенности построения кода библиотеки.

Примечание

Библиотека типов включается (или не включается) в проект по желанию разработчика. Для включения библиотеки типов в проект при создании нового объекта COM необходимо установить флажок **Include Type Library**. Иначе можно воспользоваться Репозитарием объектов Delphi, где библиотека типов доступна на странице ActiveX.

Редактор библиотеки типов предоставляет разработчику полный набор инструментов, позволяя автоматизировать процесс создания объектов и интерфейсов. Окно Редактора разделено на три основные части (см. рис. 24.5).

Вверху расположена узкая панель инструментов. При помощи кнопок панели можно создавать новые элементы COM и выполнять общие для всей библиотеки операции. Панель разделена на четыре части.

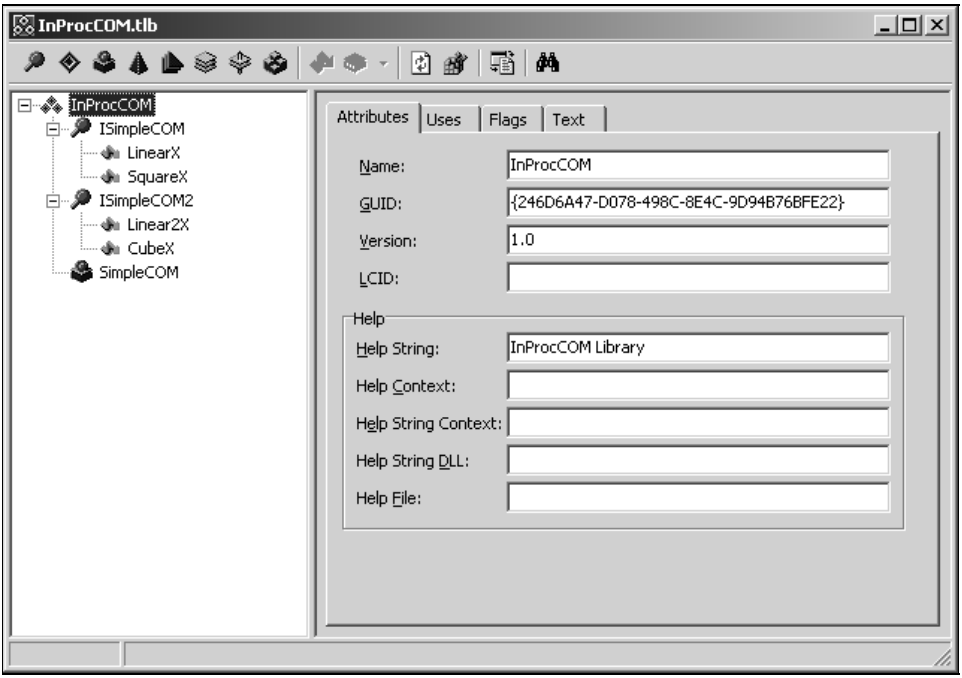


Рис. 24.5. Редактор библиотеки типов

Слева расположены кнопки новых типов, которые можно добавлять к библиотеке типов.

(Interface) — создает новый интерфейс.

(Dispatch) — создает новый диспинтерфейс *(подробнее см. главу 25)*.

(CoClass) — создает новый `CoClass` для объекта COM.

(Enum) — создает новый перечислитель. В его состав могут входить константы. Используется для инициализации объектов.

(Alias) — создает новый псевдоним (не путать с псевдонимами баз данных). Представляет собой именование существующего типа данных. Имеет собственный идентификатор.

(Record) — создает новую запись, под которой понимается список полей — именованных типов данных, не имеющих собственных идентификаторов.

(Union) — создает новое объединение, которое представляет собой индексированный упорядоченный список полей.

(Module) — создает новый модуль, который объединяет отдельные функции и константы.

Затем идут две кнопки, назначение которых меняется в зависимости от текущего типа библиотеки, выбранного в дереве слева. Эти кнопки предназначены для создания новых элементов типа. Например, для интерфейса можно создавать новые методы и переменные.

Затем следуют две группы кнопок, обеспечивающих выполнение общих операций библиотеки.

(Refresh) — обновляет исходный код библиотеки, объектов и интерфейсов в других модулях.

(Register) — проводит регистрацию библиотеки типов в системном реестре.

(Export) — экспортирует код библиотеки в формат языка Microsoft IDL или CORBA IDL.

Слева помещен иерархический список доступных в библиотеке типов частей проекта. Он используется для выбора нужного типа и управления типами — всплывающее меню содержит набор основных команд редактора.

Справа расположена информационная панель, которая отображает сведения о выбранном в списке элементе и позволяет управлять его параметрами. Ее многостраничный блокнот изменяет состав и содержимое страниц в зависимости от текущего типа.

Для любого типа в панели всегда присутствуют две страницы. Страница **Attributes** задает основные параметры типа. Страница **Text** может содержать текстовое описание типа.

Пример работы с библиотекой типов рассматривается далее.

Простой объект COM в составе внутреннего сервера

Теперь рассмотрим небольшой пример, который несмотря на простоту, охватывает все основные этапы работы с объектами COM.

Для демонстрации работы объекта COM создадим внутренний сервер, а также простой объект COM в составе внутреннего сервера, предоставляю-

щего собственный интерфейс. Интерфейс будет содержать несколько методов, выполняющих простейшие математические операции.

Создание объекта

Для создания объекта COM в составе внутреннего сервера (динамической библиотеки) необходимо выполнить следующие действия.

В категории **Delphi Projects\ActiveX** Репозитория Delphi требуется выбрать значок **ActiveX Library**. В результате будет создан новый проект, который мы назовем `ClientInProcCOM`. Исходный код новой динамической библиотеки будет создан автоматически. Он представлен в листинге 24.1.

Листинг 24.1. Исходный код динамической библиотеки `InProcCOM`

```
library InProcCOM;
uses
  ComServ,
  ClientInProcCOM_TLB in 'ClientInProcCOM_TLB.pas',
  uSimpleCOM in 'uSimpleCOM.pas' {SImpleCOM: CoClass};
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
begin
end.
```

Четыре экспортируемые библиотекой функции предназначены для обеспечения ее взаимодействия с COM:

- `DllGetClassObject` — обеспечивает доступ к библиотеке классов;
- `DllCanUnloadNow` — управляет работой сервера в зависимости от состояния объектов;
- `DllRegisterServer` — регистрирует сервер в системном реестре;
- `DllUnregisterServer` — удаляет информацию о сервере из системного реестра.

Затем из Репозитория необходимо выбрать значок **COM Object** в категории **Delphi Projects | ActiveX**. После щелчка на значке открывается диалог установки начальных параметров нового объекта (рис. 24.6).

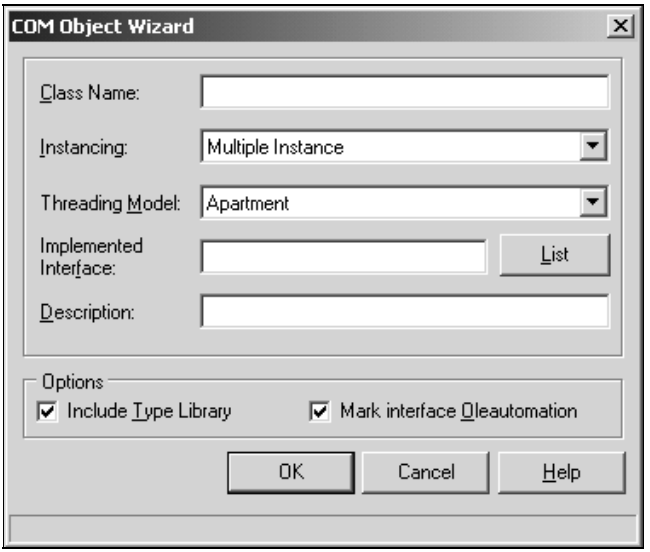


Рис. 24.6. Диалог установки начальных параметров объекта COM

Однострочный редактор **Class Name** должен содержать имя нового класса.

Комбинированный список **Instancing** определяет способ создания объекта:

- **Internal** — объект используется в процессе;
- **Single Instance** — при обращении к объекту нескольких клиентов в единственном экземпляре сервера создается необходимое число объектов;
- **Multiple Instance** — если к объекту обращается несколько клиентов, то для каждого создается собственный экземпляр сервера объекта.

При использовании объекта внутри процесса установки в этом списке не имеют значения.

Комбинированный список **Threading Model** определяет способ взаимодействия объекта и клиентов:

- **Single** — сервер может обслуживать вызовы клиентов только последовательно, один за другим;
- **Apartment** — вызов объекта осуществляется только в одном потоке, в котором создан и сам объект COM, при этом объект может обслуживать только одного клиента одновременно. В сервере в таком режиме одновременно могут работать несколько объектов;
- **Free** — объект может обслуживать произвольное число клиентов одновременно;
- **Both** — объект может работать с клиентами в моделях Apartment и Free;

- **Neutral** — объект может обслуживать несколько клиентов в различных нитях, но при этом не разрешает возникающие конфликты. Применяется только для технологии COM+.

Однострочный редактор **Implemented Interfaces** содержит имя интерфейса, инкапсулированного создаваемым объектом. По умолчанию для объекта создается собственный интерфейс, имя которого состоит из имени объекта и префикса `I`. Однако разработчик может назначить объекту и один из уже существующих интерфейсов COM. Вы можете задать нужный интерфейс вручную или выбрать его из диалога **Interface Selection Wizard** который открывается при нажатии кнопки **List** (рис. 24.7).

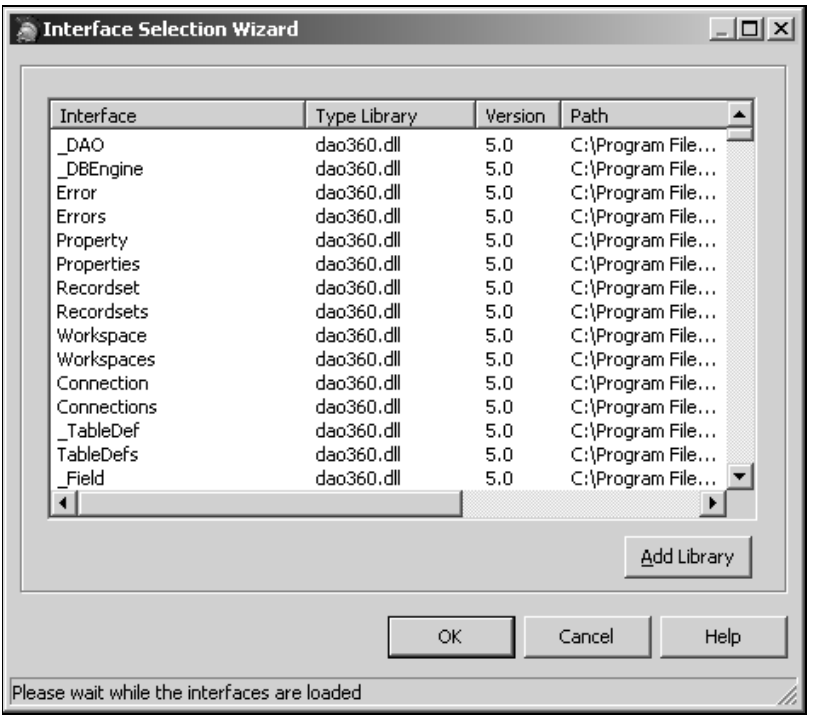


Рис. 24.7. Диалог **Interface Selection Wizard**

Однострочный редактор **Description** содержит описание объекта.

Флажок **Include Type Library** управляет созданием библиотеки типов для объекта. Однако он недоступен, если вы выберете для объекта уже существующий интерфейс.

Флажок **Mark interface Oleautomation** делает объект пригодным для технологии Автоматизации. Если флажок включен, для интерфейса объекта создается диспинтерфейс *(см. главу 25)*.

Настроенные параметры объекта `TSimpleCOM` представлены в табл. 24.1.

Таблица 24.1. Параметры объекта `TSimpleCOM`

Параметр	Значение
Class Name	SimpleCOM
Instancing	tmApartment
Threading Model	ciMultiInstance
Implemented Interfaces	ISimpleCOM
Description	Simple COM Object Sample
Include Type Library	Включен
Mark interface Oleautomation	Выключен

После установки параметров и щелчка на кнопке **OK** к проекту будет добавлен новый модуль с исходным кодом нового класса.

Исходный код модуля объекта COM сразу после создания представлен в листинге 24.2.

Листинг 24.2. Исходный код объекта COM `TSimpleCOM` сразу после создания

```
unit uSimpleCOM;
{$WARN SYMBOL_PLATFORM OFF}
interface
uses
  Windows, ActiveX, Classes, ComObj, ClientInProcCOM_TLB, StdVcl;
type
  TSImpleCOM = class(TTypedComObject, ISImpleCOM)
  protected
  end;

implementation
uses ComServ;
initialization
  TTypedComObjectFactory.Create(ComServer, TSImpleCOM, Class_SImpleCOM,
    ciMultiInstance, tmApartment);
end.
```

Как видно из объявления, ближайшим предком класса нового объекта является класс `TTypedComObject`, т. к. в соответствии с исходными настройками объекта для него создана фабрика класса.

Конструктор объекта `TSimpleCOM` содержит имя фабрики класса и базовый интерфейс, имя которого совпадает с именем класса. В секции инициализации расположен конструктор фабрики класса. Обратите внимание, что класс фабрики класса был создан автоматически.

Работу сервера обеспечивают модуль `ComServ` и переменная `ComServ`, указывающая на класс сервера COM. Переменная содержится в конструкторе фабрики класса.

Как видите, в модуле объекта COM описан только класс `TSimpleCOM`, который является основой функциональности объекта COM. Интерфейс `ISimpleCOM` и `CoClass` объявлены в библиотеке типов в файле ClientInProcCOM_TLB.PAS.

Исходный код библиотеки был создан автоматически вместе с новым объектом COM и сохранен в файле ClientInProcCOM_TLB.PAS. Листинг 24.3 содержит исходный код библиотеки типов для объекта COM `TSimpleCOM`.

Листинг 24.3. Исходный код библиотеки типов для объекта COM `TSimpleCOM`

```
unit InProcCOM_TLB;
//
************************************************************************
//
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or
the
// 'Refresh' command of the Type Library Editor activated while editing
the
// Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost.
//
************************************************************************
//
// PASTLWTR : $Revision:   1.130.3.0.1.0  $
// File generated on 11.03.2002 17:46:03 from Type Library described
below.
//
************************************************************************
//
```

```
// Type Lib: C:\Mark\Bin\Books\d6WEB\DemosE\01_1\Server\InProcCOM.tlb (1)
// LIBID: {246D6A47-D078-498C-8E4C-9D94B76BFE22}
// LCID: 0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINNT\System32\stdole2.tlb)
//   (2) v4.0 StdVCL, (C:\Program Files\Common
     Files\SuperCollect\stdvcl40.dll)
//
************************************************************************
//
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked point-
ers.
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}


interface
uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;
//
************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
//
************************************************************************//
const
  // TypeLibrary Major and minor versions
  InProcCOMMajorVersion = 1;
  InProcCOMMinorVersion = 0;

  LIBID_InProcCOM: TGUID = '{246D6A47-D078-498C-8E4C-9D94B76BFE22}';

  IID_ISimpleCOM: TGUID = '{76247090-8BCD-4AAD-8EE3-EEE4DE68872C}';
  CLASS_SimpleCOM: TGUID = '{E76AE0BE-14DD-431D-B923-324EBA987770}';
type
//
************************************************************************//
// Forward declaration of types defined in TypeLibrary
```

```
//
*********************************************************************//
  ISimpleCOM = interface;
//
*********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
//
*********************************************************************//
  SimpleCOM = ISimpleCOM;
//
*********************************************************************//
// Interface: ISimpleCOM
// Flags:     (0)
// GUID:      {76247090-8BCD-4AAD-8EE3-EEE4DE68872C}
//
*********************************************************************//
  ISimpleCOM = interface(IUnknown)
    ['{76247090-8BCD-4AAD-8EE3-EEE4DE68872C}']
  end;
//
*********************************************************************//
// The Class CoSimpleCOM provides a Create and CreateRemote method to
// create instances of the default interface ISimpleCOM exposed by
// the CoClass SimpleCOM. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
//
*********************************************************************//
  CoSimpleCOM = class
    class function Create: ISimpleCOM;
    class function CreateRemote(const MachineName: string): ISimpleCOM;
  end;
implementation
uses ComObj;
class function CoSimpleCOM.Create: ISimpleCOM;
begin
  Result := CreateComObject(CLASS_SimpleCOM) as ISimpleCOM;
end;
class function CoSimpleCOM.CreateRemote(const MachineName: string):
ISimpleCOM;
```

```
begin
  Result := CreateRemoteComObject(MachineName, CLASS_SimpleCOM) as ISim-
pleCOM;
end;
end.
```

Библиотека типов содержит автоматически сгенерированный `GUID` интерфейса `ISimpleCOM`, затем идет объявление самого интерфейса. Класс `CoSimpleCOM` обеспечивает работу интерфейсов объекта `SimpleCOM`. Пока он содержит только один интерфейс. В этом классе созданы две функции: `Create` — для работы в сервере в процессе разработки приложения и в локальном сервере, `CreateRemote` — для работы в удаленном сервере. Обе функции возвращают указатели на интерфейс `ISimpleCOM`.

Создание методов интерфейса

Создав новый объект COM, займемся разработкой методов, которые реализуют его функции. Предположим, что объект предназначен для расчета значений некоторых классов простейших математических функций.

Нам необходимо показать использование нескольких интерфейсов одного объекта, иначе объект COM принципиально не будет отличаться от обычного объекта. Пусть первый и второй интерфейсы реализуют различные простейшие линейные и степенные функции.

Первый интерфейс `ISimpleCOM` у нас уже имеется, он был создан вместе с объектом COM на предыдущем этапе.

Для создания второго интерфейса и всех необходимых методов воспользуемся Редактором библиотеки типов (см. рис. 24.5). Для этого требуется выполнить следующие действия.

1. В иерархическом списке необходимо выбрать интерфейс `ISimpleCOM` и щелкнуть на кнопке **New Method** в панели инструментов (рис. 24.8).
2. Появившийся в результате в иерархическом списке метод `Method1` нужно переименовать в `LinearX`. Обратите внимание, что для методов на странице **Attributes** можно задать только имя.
3. Затем необходимо перейти на страницу **Parameters** в правой части редактора и в списке **Return Type** задать тип `long` — тип возвращаемого методом результата вместо стандартного типа `HResult`.
4. Теперь щелкнуть на кнопке **Add** в нижней части страницы, чтобы добавить методу первый параметр. В появившейся строке в ячейке **Name** следует изменить имя параметра на `AValue` (рис. 24.9).

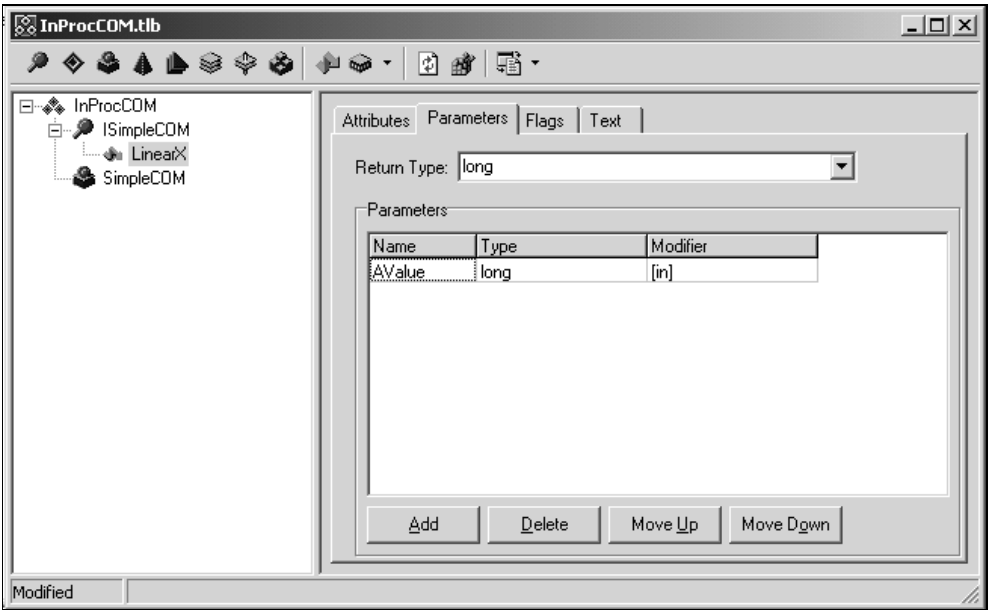


Рис. 24.8. Создание нового метода в Редакторе библиотеки типов

5. Тип данных параметра задается в колонке **Type** (см. рис. 24.9). Выпадающий список колонки содержит все допустимые типы данных. Для нашего параметра нужно выбрать тип `long`.
6. Тип самого параметра задается в колонке **Modifier**. При щелчке на кнопке появляется окно **Parameter Flags** (рис. 24.10), в котором можно задать тип параметра, а также при необходимости и значение по умолчанию. В нашем случае параметр должен быть входящим, поэтому в окне должен быть включен флажок **In**.

После выполнения этих операций объявление метода готово.

Проделаем ту же последовательность действий для еще одного метода — `SquareX`.

Для того чтобы создать исходный код для этих методов, необходимо щелкнуть на кнопке **Refresh** в панели инструментов Редактора библиотеки типов. В результате в модуле объекта `SimpleCOM` автоматически появится исходный код для созданных методов. Остается только ввести в них соответствующие математические функции (см. листинг 24.4).

Для создания второго интерфейса и всех необходимых методов также воспользуемся Редактором библиотеки типов. Для этого требуется выполнить следующие действия.

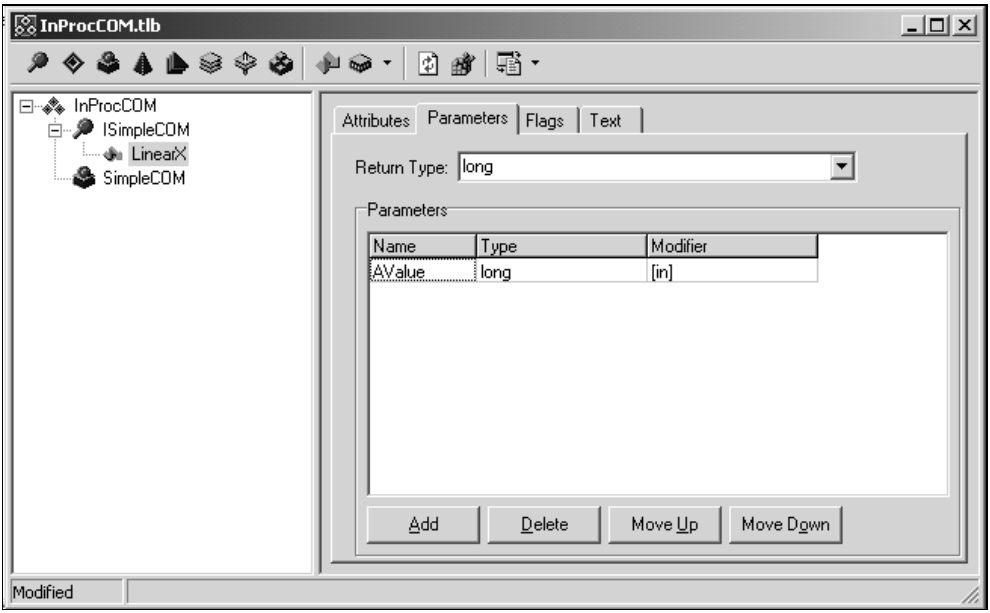


Рис. 24.9. Настройка параметров метода

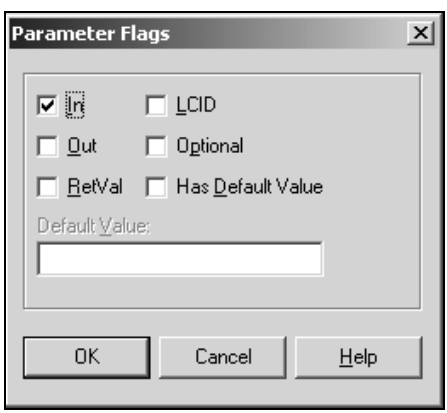


Рис. 24.10. Окно **Parameter Flags**

Для создания второго интерфейса необходимо щелкнуть на кнопке **Interface** в панели инструментов. Появившийся в иерархическом списке новый интерфейс нужно переименовать, присвоив ему имя `ISimpleCOM2`.

На странице **Attributes** (рис. 24.11) для нового интерфейса генерируется `GUID`, а в списке **Parent Interface** требуется выбрать `IUnknown` — имя интерфейса-предка. Помимо этого, на странице **Flags** следует отключить флажки

Dual и **OleAutomation**. Для вновь создаваемых интерфейсов эти флажки включены по умолчанию, т. к. за основу внутреннего сервера мы взяли динамическую библиотеку ActiveX *(см. главу 26)*.

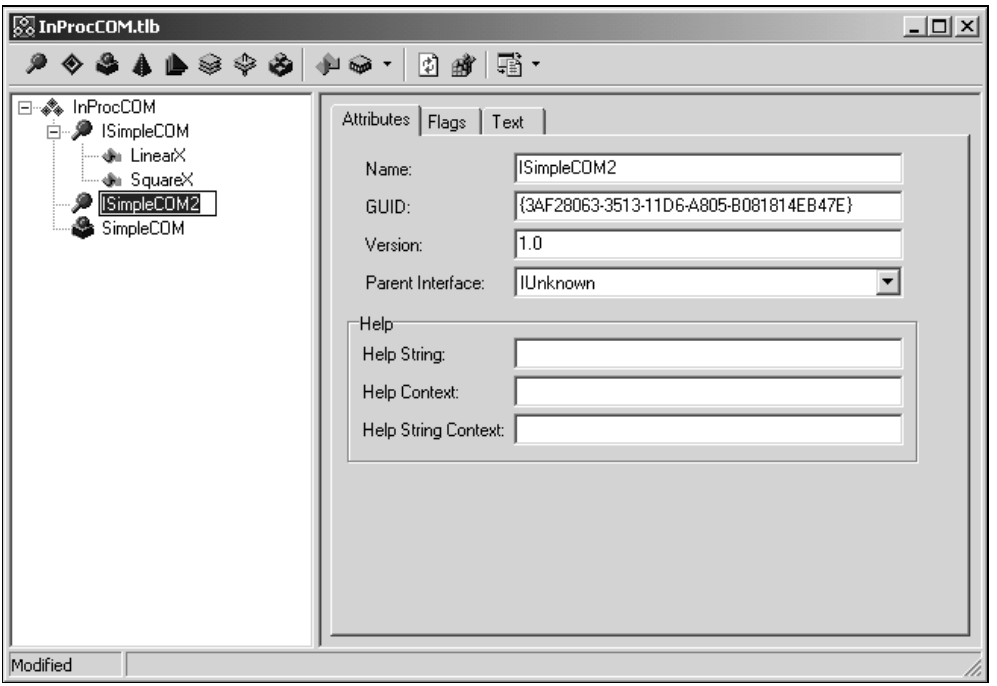


Рис. 24.11. Создание нового интерфейса

Затем в соответствии с описанными операциями для нового интерфейса `ISimpleCOM2` создаются методы `Linear2X` и `CubeX`.

Теперь созданный интерфейс требуется привязать к объекту COM `TSimpleCOM`. Для этого в иерархическом списке выбирается объект, а в правой части — страница **Implements**. На ней расположен список всех интерфейсов объекта, пока в нем только один базовый интерфейс (рис. 24.12). Щелкните на списке правой кнопкой мыши и из всплывающего меню выберите команду **Insert Interface**. Из появившегося списка выберите интерфейс `ISimpleCOM2` и щелкните на кнопке **OK** — новый интерфейс появится в объекте.

В завершение щелкните на кнопке **Refresh** для создания соответствующего исходного кода. Затем можно перейти в модуль uSimpleCOM.PAS и создать простейший исходный код для новых методов (см. листинг 24.4). Каждый из методов выполняет простую арифметическую операцию.

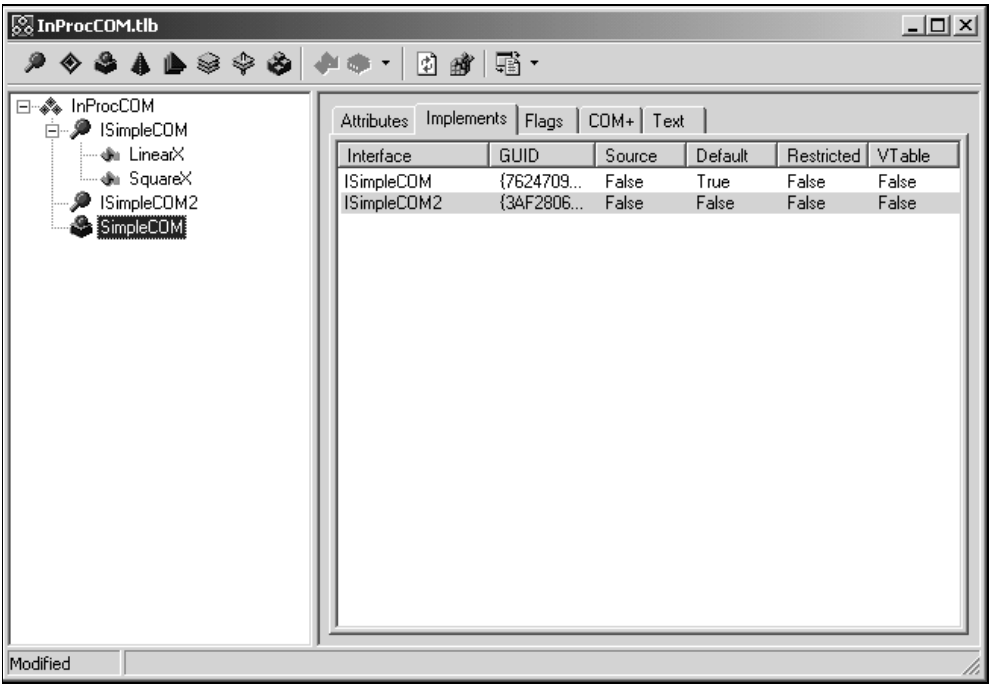


Рис. 24.12. Связывание интерфейса с объектом COM

Листинг 24.4. Модуль `uSimpleCOM` объекта `SimpleCOM` после создания второго интерфейса

```
unit uSimpleCOM;
{$WARN SYMBOL_PLATFORM OFF}
interface
uses
  Windows, ActiveX, Classes, ComObj, InProcCOM_TLB, StdVcl;
type
  TSimpleCOM = class(TTypedComObject, ISimpleCOM, ISimpleCOM2)
  protected
    function LinearX(AValue: Integer): Integer; stdcall;
    function SquareX(AValue: Integer): Integer; stdcall;
    function CubeX(AValue: Integer): Integer; stdcall;
    function Linear2X(AValue: Integer): Integer; stdcall;
  end;
```

```
implementation
uses ComServ;

function TSimpleCOM.LinearX(AValue: Integer): Integer;
begin
 Result := AValue;
end;
function TSimpleCOM.SquareX(AValue: Integer): Integer;
begin
 Result := AValue*AValue;
end;
function TSimpleCOM.CubeX(AValue: Integer): Integer;
begin
 Result := AValue*AValue*AValue;
end;
function TSimpleCOM.Linear2X(AValue: Integer): Integer;
begin
 Result := AValue*2;
end;
initialization
  TTypedComObjectFactory.Create(ComServer, TSimpleCOM, Class_SimpleCOM,
    ciMultiInstance, tmApartment);
end.
```

Обратите внимание, что в представленном коде нет даже намека на принадлежность методов к разным интерфейсам. Это разделение осуществляется в библиотеке типов (листинг 24.5). Зато в объявлении класса `TSimpleCOM` появился второй интерфейс `ISimpleCOM2`.

Листинг 24.5. Исходный код библиотеки типов после создания второго интерфейса (комментарии удалены)

```
unit InProcCOM_TLB;
{$TYPEDADDRESS OFF}
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}
interface
uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;
const
  InProcCOMMajorVersion = 1;
  InProcCOMMinorVersion = 0;
```

```
  LIBID_InProcCOM: TGUID = '{246D6A47-D078-498C-8E4C-9D94B76BFE22}';
  IID_ISimpleCOM: TGUID = '{76247090-8BCD-4AAD-8EE3-EEE4DE68872C}';
  IID_ISimpleCOM2: TGUID = '{3AF28063-3513-11D6-A805-B081814EB47E}';
  CLASS_SimpleCOM: TGUID = '{E76AE0BE-14DD-431D-B923-324EBA987770}';
type
  ISimpleCOM = interface;
  ISimpleCOM2 = interface;
  SimpleCOM = ISimpleCOM;
  ISimpleCOM = interface(IUnknown)
    ['{76247090-8BCD-4AAD-8EE3-EEE4DE68872C}']
    function LinearX(AValue: Integer): Integer; stdcall;
    function SquareX(AValue: Integer): Integer; stdcall;
  end;
  ISimpleCOM2 = interface(IUnknown)
    ['{3AF28063-3513-11D6-A805-B081814EB47E}']
    function Linear2X(AValue: Integer): Integer; stdcall;
    function CubeX(AValue: Integer): Integer; stdcall;
  end;
  CoSimpleCOM = class
    class function Create: ISimpleCOM;
    class function CreateRemote(const MachineName: string): ISimpleCOM;
  end;
implementation
uses ComObj;
class function CoSimpleCOM.Create: ISimpleCOM;
begin
  Result := CreateComObject(CLASS_SimpleCOM) as ISimpleCOM;
end;
class function CoSimpleCOM.CreateRemote(const MachineName: string):
ISimpleCOM;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_SimpleCOM) as
ISimpleCOM;
end;
end.
```

Рассмотрим теперь изменения в библиотеке типов.

Во-первых, в библиотеке появилось объявление второго интерфейса `ISimpleCOM2`. Для него сгенерирован `GUID` и объявлены необходимые переменные.

Во-вторых, созданные в Редакторе библиотеки типов методы объявлены в соответствующих интерфейсах. Обратите внимание, что заданный для параметров методов в Редакторе библиотеки типов тип данных `long` был преобразован в тип `integer`.

Регистрация внутреннего сервера

После завершения разработки объекта и компиляции библиотеки необходимо зарегистрировать ее в качестве сервера. Для этого используется команда **Register ActiveX Server** меню **Run** Главного окна Delphi. При этом информация о внутреннем сервере помещается в системном реестре.

При создании объекта COM `TSimpleCOM` динамическая библиотека `InProcCOM` подгружается в адресное пространство клиентского приложения и работает как внутренний сервер. В результате, если клиентское приложение имеет ссылки на интерфейсы объекта COM, то оно может задействовать методы этих интерфейсов.

На этом создание объекта `SimpleCOM` закончено, можно переходить к клиентской части проекта.

Использование интерфейсов внутреннего сервера COM

Роль клиентского сыграет простое приложение `ClientInProcCOM` с единственной формой `fmMain`. Необходимо настроить ее таким образом, чтобы можно было протестировать работу сразу двух интерфейсов созданного объекта (рис. 24.13).

Компоненты `TSpinEdit` предназначены для задания значения X для функций, реализованных в методах интерфейсов `ISimpleCOM` и `ISimpleCOM2`. Значения из этих визуальных компонентов будут переданы методам в качестве параметров.

Кнопки должны вызывать методы интерфейсов объекта. Для отображения результата предусмотрены компоненты `TLabel`.

Для того чтобы клиентское приложение могло работать с интерфейсами внутреннего сервера `InProcCOM`, необходимо добавить в проект клиента файл библиотеки типов InProcCOM_TLB.PAS.

После этого можно объявить две переменные для интерфейсов объекта COM `TSimpleCOM`:

```
Interface1: ISimpleCOM;
Interface2: ISimpleCOM2;
```

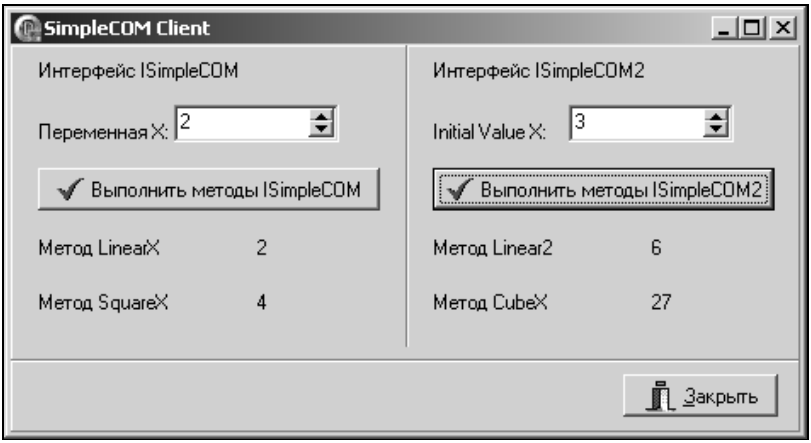


Рис. 24.13. Главная форма проекта ClientInProcCOM

В листинге 24.6 представлен исходный код клиента, обеспечивающий использование методов объекта COM.

Листинг 24.6. Секция `implementation` модуля `uMain` проекта ClientInProcCOM

```
implementation
uses InProcCOM_TLB;
var
  Interface1: ISimpleCOM;
  Interface2: ISimpleCOM2;
{$R *.dfm}
procedure TfmMain.FormShow(Sender: TObject);
begin
 Interface1 := CoSimpleCOM.Create;
 Interface1.QueryInterface(ISimpleCOM2, Interface2);
end;
procedure TfmMain.bbRunSimpleCOMClick(Sender: TObject);
begin
 laLinearRes.Caption :=
 IntToStr(Interface1.LinearX(seSimpleCOMValue.Value));
 laSquareRes.Caption :=
 IntToStr(Interface1.SquareX(seSimpleCOMValue.Value));
end;
procedure TfmMain.bbRunSimpleCOM2Click(Sender: TObject);
```

```
begin
 laLinear2Res.Caption :=
IntToStr(Interface2.Linear2X(seSimpleCOM2Value.Value));
 laCubeRes.Caption :=
IntToStr(Interface2.CubeX(seSimpleCOM2Value.Value));
end;
end.
```

При открытии формы в методе-обработчике `FormShow` создается `CoClass` класса `TSimpleCOM` — `CoSimpleCOM`. Его конструктор возвращает указатель на базовый интерфейс `ISimpleCOM` в переменную `Interface1` типа `ISimpleCOM`. Все эти операции выполняются динамической библиотекой `InProcCOM`, запись которой в системном реестре можно найти по глобальному идентификатору из библиотеки типов InProcCOM_TLB.PAS.

Так как любой интерфейс является наследником `IUnknown`, то для получения указателя на второй интерфейс можно воспользоваться методом `QueryInterface`, в параметрах которого указывается идентификатор нужного интерфейса и переменная, в которую будет возвращен указатель. В нашем примере это интерфейс `ISimpleCOM2` и переменная `Interface2`.

После выполнения этих операций клиент получает указатели на оба интерфейса объекта `SimpleCOM`. Для выполнения методов этих интерфейсов использованы методы-обработчики нажатия кнопок `bbRunSimpleCOM` и `bbRunSimpleCOM2`.

Результат отображается на форме присвоением результатов работы методов интерфейсов объекта COM `TSimpleCOM` внутреннего сервера InProcCOM.DLL.

Резюме

Технология COM является базовой для тем, рассматриваемых в этой книге. С ее помощью могут взаимодействовать объекты, созданные при помощи различных средств программирования на различных языках программирования.

Краеугольными понятиями COM являются объект, интерфейс и фабрика класса.

Интерфейс объединяет несколько методов и обеспечивает доступ к ним по общему указателю.

Объект инкапсулирует один или несколько интерфейсов и предоставляет их клиентам.

Фабрика класса создает экземпляры объектов COM.

Глава 25

Технология Автоматизация

На основе технологии COM, рассмотренной *в главе 24*, разработаны многие другие технологии. Одна из них — Автоматизация (Automation) — была создана довольно давно и широко распространена в различных пользовательских приложениях ОС Windows.

Эта технология обеспечивает использование функций одних приложений из других приложений. Взаимодействие между ними осуществляется за счет набора специально созданных интерфейсов. Основополагающим является интерфейс `IDispatch`.

Автоматизацию поддерживают такие известные программы, как Word, Excel и Internet Explorer. О преимуществах этой технологии говорить излишне. Например, любая пользовательская программа, имеющая информацию о функциях, предоставляемых интерфейсом Автоматизации Microsoft Word, получает доступ к обширным возможностям редактирования текстов.

С точки зрения программиста, создание приложений на основе технологии Автоматизации мало чем отличается от методики, рассмотренной в *главе 24*, для объектов COM. В этой главе рассматриваются особенности программирования в Delphi с использованием Автоматизации. Среди них выделим следующие вопросы.

- Интерфейсы автоматизации.
- Объекты автоматизации.
- Создание сервера автоматизации.
- Создание контроллера автоматизации.
- Примеры автоматизации.

Базовые понятия Автоматизации

Автоматизация, как технология, основанная на COM, обеспечивает предоставление функций одних приложений другим. Естественно, что это делается за счет интерфейсов. Эти интерфейсы инкапсулированы разнообразными

объектами Автоматизации, которые, в свою очередь, входят в состав серверов Автоматизации.

Основным отличием Автоматизации от родительской технологии COM является способ вызова методов интерфейсов.

В COM указатели на методы интерфейсов содержатся исключительно в специальных виртуальных таблицах (см. рис. 24.3). Такой способ доступа называется *ранним связыванием*.

В Автоматизации доступ к методам интерфейсов осуществляется как традиционным способом COM, так и при помощи специального метода, который ищет запрошенный метод по специальному идентификатору. Такой способ доступа называется *поздним связыванием*.

В остальном же технология Автоматизация подобна технологии COM. Рассмотрим основные сущности, используемые в Автоматизации.

Базовый интерфейс Автоматизации — `IDispatch` — вполне обычный интерфейс COM, обладающий, тем не менее, несколькими очень важными для Автоматизации методами. Как и все подобные интерфейсы, он реализован при помощи виртуальной таблицы *(см. главу 24)* указателей на его методы.

Виды интерфейсов Автоматизации

Все интерфейсы Автоматизации имеют общего предка — интерфейс `IDispatch`, который и обеспечивает уникальную функциональность технологии. И конечно же общим предком любого интерфейса Автоматизации является интерфейс `IUnknown` *(см. главу 24)*.

Рассматриваемые здесь интерфейсы можно разделить на две группы: диспинтерфейсы и дуальные интерфейсы.

Диспинтерфейсы

Особенность интерфейса `IDispatch` заключается в наличии специального метода `Invoke`, который обеспечивает вызов других методов. Кроме указанного метода `IDispatch` имеет еще три важных метода: `GetTypeInfoCount`, `GetTypeInfo` и `GetIDsOfNames`. Все они обеспечивают механизм позднего связывания.

Объект, работающий с использованием интерфейса `IDispath`, должен обязательно определить дополнительный интерфейс, в котором перечисляются методы, доступные для вызова через метод `Invoke`. Такой интерфейс называется *интерфейсом диспетчеризации* или *диспинтерфейсом*.

Вместо виртуальной таблицы для доступа к методам в нем применяется метод `Invoke`. Реализация метода `Invoke` по сути является оператором `case`: в

нем на основе переданного в параметре идентификатора метода просто выбирается нужный метод.

При таком вызове каждому методу должен соответствовать уникальный идентификатор, который называется *диспетчерским идентификатором* (DISPID).

Методы диспинтерфейсов имеют некоторые ограничения на типы данных параметров; так, при вызове методов значения всех параметров конвертируются в тип "вариант". При возвращении результатов выполняется обратная процедура.

Рассмотрим механизм позднего связывания (рис. 25.1).

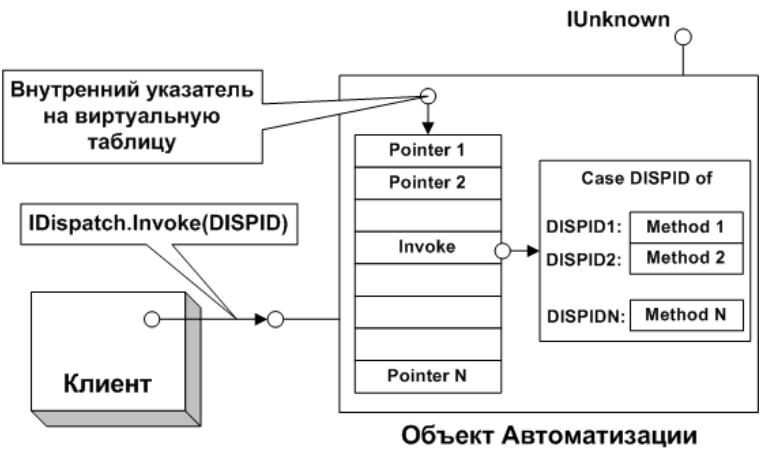


Рис. 25.1. Механизм вызова методов диспинтерфейса

Когда клиенту необходим тот или иной метод интерфейса Автоматизации, он вызывает метод `Invoke`, передавая ему в качестве параметра диспетчерский идентификатор нужного метода. Далее объект Автоматизации по своей виртуальной таблице находит указатель на метод `Invoke` и вызывает его. Метод `Invoke` реализует поиск нужного метода по его диспетчерскому идентификатору. Найденный указатель возвращается основному интерфейсу, который использует его для обработки запроса клиента.

Каждый диспинтерфейс, как и любой другой интерфейс, имеет собственный уникальный идентификатор. Он необходим в случае, если в приложении имеется несколько диспинтерфейсов. Так как интерфейс `IDispatch` является наследником `IUnknown`, то для вызова другого диспинтерфейса можно воспользоваться унаследованным методом `QueryInterface`.

В результате взаимодействие между приложениями Автоматизации упрощается — ведь для успешной работы требуется всего одна виртуальная таблица

для интерфейса IDispatch, все остальные методы вызываются описанным выше механизмом.

Диспинтерфейсы имеют еще одно преимущество. Для них можно напрямую определять методы, призванные возвращать и задавать значения свойств. Метод, возвращающий значение свойства, выполняет только чтение. Метод, задающий значение, выполняет только запись.

Дуальные интерфейсы

Однако диспинтерфейсы работают несколько медленнее стандартных. Поэтому в приложениях Автоматизации существует еще один тип интерфейсов — *дуальные интерфейсы*. Его методы можно вызывать как через Invoke, так и через виртуальную таблицу.

Дуальный интерфейс обязательно должен быть наследником IDispatch. Его виртуальная таблица включает ссылки на три метода IUnknown, четыре метода IDispatch и методы соответствующего диспинтерфейса.

Библиотека типов

Так как при вызове методов диспинтерфейса клиент должен обладать информацией о диспетчерских идентификаторах вызываемых методов, при создании интерфейсов Автоматизации значительную роль играет Библиотека типов *(см. главу 24)*. Предоставленные библиотекой типов идентификаторы методов, а также информация о параметрах методов и самих интерфейсов используются клиентом при вызове метода Invoke интерфейса IDispatch.

Маршалинг интерфейсов Автоматизации

Для обеспечения работы локальных и удаленных серверов Автоматизации также применяется механизм маршалинга и демаршалинга. Маршалинг отвечает за упаковку параметров запроса, демаршалинг обеспчеивает распаковку *(см. главу 24)*.

Как и для любого интерфейса с виртуальной таблицей методов, интерфейс IDispatch и его потомки для маршалинга и демаршалинга нуждаются в двух вспомогательных объектах — заместителе и заглушке *(см. главу 24)*. Но при помощи этой пары могут быть обработаны параметры методов только одного интерфейса. Однако методы диспинтерфейса, вызываемые методом Invoke, могут иметь самые разнообразные параметры, отличные от параметров методов базового интерфейса.

В этом случае дополнительную обработку выполняет сам метод Invoke, преобразуя параметры в переменные типа "вариант". И поэтому маршалинг диспинтефрейсов имеет особенность: при передаче разнотипных параметров

через метод `Invoke` происходит конвертация параметров в тип "вариант" и обратно. Поэтому методы интерфейсов Автоматизации работают с ограниченным набором типов данных, пригодных для конвертации через вариант.

Именно потому, что при маршалинге диспинтерфейсов не требуются заместитель и заглушка, диспинтерфейсы поддерживают позднее связывание.

Объект Автоматизации

В рамках технологии Автоматизации (так же как и в COM) все методы инкапсулированы в *объекте Автоматизации*, который может содержать один или более интерфейсов. Подобно объекту COM, объект Автоматизации работает в составе сервера Автоматизации и предоставляет клиенту методы своих интерфейсов. Все интерфейсы объекта Автоматизации должны быть интерфейсами диспетчеризации.

Сервер Автоматизации

Приложение, предоставляющее свои функции при помощи интерфейса `IDispatch`, называются *сервером Автоматизации*. Сервер обязательно должен включать *объект Автоматизации* — обычный объект COM, имеющий интерфейс `IDispatch` и диспинтерфейсы.

Сервер Автоматизации представляет собой исполняемый файл (приложение или динамическую библиотеку), который может содержать один или несколько объектов одного или разных классов. Серверы Автоматизации (как и COM) могут быть трех типов: внутренним, локальным и удаленным.

Контроллер Автоматизации

Любой стандартный клиент COM, который обращается к серверу Автоматизации через `IDispatch`, называется *контроллером Автоматизации*.

Реализация Автоматизации в Delphi

Теперь давайте посмотрим, как технология Автоматизация реализована в Delphi. Основой любой программной реализации Автоматизации в Delphi является класс `TAutoObject` — потомок класса `TComObject`. Он предоставляет для использования методы интерфейса `IDispatch`. Соответственно объекты Автоматизации обладают всеми возможностями объектов COM Delphi.

При разработке приложений Автоматизации учитывается информация о типах. Библиотека типов Автоматизации создается так же, как и для обычных приложений COM.

Интерфейсы Автоматизации

Как уже упоминалось, все интерфейсы Delphi для Автоматизации (как стандартные, так и созданные разработчиками) имеют общего предка — интерфейс `IDispatch`. Для них помимо обычных методов можно объявлять и свойства с методами чтения и записи. Эта возможность реализована в Редакторе Библиотеки типов.

Интерфейс *IDispatch*

Важнейший для технологии Автоматизации интерфейс `IDispatch` объявлен в модуле System.pas:

```
IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
    HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer):
HResult; stdcall;
  end;
```

Помимо унаследованных от интерфейса `IUnknown` он содержит четыре метода.

Главный метод интерфейса

```
function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): HResult;
stdcall;
```

служит для вызова всех методов диспинтерфейса. Для этого в параметре `DispID` необходимо передать диспетчерский идентификатор вызываемого метода. `GUID` интерфейса, метод которого вызывается, определяется параметром `IID`. Параметр `LocaleID` задает локализацию передаваемых значений. Параметр `Flags` задает, вызывается ли обычный метод, или это метод чтения или записи свойства. Параметр `Params` содержит указатель на массив типа `TDispParams`, содержащий параметры вызываемого метода. Параметр `VarResult` возвращает значение, переданное назад вызываемым методом.

Если при вызове метода возникла ошибка, параметр `ExcepInfo` содержит информацию об этом. А в параметре `ArgErr` содержится индекс неверно заданного параметра из массива `Params`.

Метод

```
function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
```

сообщает о том, может ли данный объект возвращать информацию о типе во время выполнения. В параметре `Count` возвращается число интерфейсов объекта, поддерживающих информацию о типе.

Метод

```
function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HResult;
stdcall;
```

возвращает указатель на интерфейс библиотеки типов `ITypeInfo` (если он есть). Если метод выполнен успешно, параметр `TypeInfo` содержит указатель на структуру `ITypeInfo`, содержащую информацию о типе.

Метод

```
function GetIDsOfNames(const IID: TGUID; Names: Pointer; NameCount,
LocaleID: Integer; DispIDs: Pointer): Hresult; stdcall;
```

возвращает диспетчерский идентификатор по заданному имени метода диспинтефрейса. Массив имен методов передается в параметре `Names` с числом элементов `NameCount`. Указатель `DispIDs` определяет массив идентификаторов.

Интерфейсы диспетчеризации

При создании полноценных серверов Автоматизации, к которым могут обращаться контроллеры, не имеющие полной информации о функциях сервера, необходимо реализовывать интерфейсы диспетчеризации. Они являются наследниками интерфейса `IDispatch`, но при их объявлении употребляется ключевое слово `dispinterface`:

Например:

```
ISomeDisp = dispinterface
 ['{AEE41D32-9FFE-94D0-4395-1120AA74DE39}']
 function Method1: OleVariant; dispid 1;
 function Method2: OleVariant; dispid 2;
 procedure Method3; dispid 3;
 procedure Method4; dispid 4;
end;
```

Каждый метод интерфейса диспетчеризации должен иметь уникальный идентификатор `dispid`. Свойства должны иметь атрибут только для чтения или только для записи. Параметры и возвращаемые результаты методов должны иметь только такие типы: `Byte`, `Currency`, `Real`, `Double`, `Real48`, `Integer`, `Single`, `Smallint`, `AnsiString`, `ShortString`, `TDateTime`, `Variant`, `OleVariant` и `WordBool`.

Для обращения к методам и свойствам интерфейсов диспетчеризации применяются методы `GetIDsOfNames` и `Invoke`. Интерфейсы диспетчеризации работают на основе позднего связывания.

Дуальные интерфейсы

Реализуемые в приложениях Автоматизации Delphi дуальные интерфейсы создаются на базе `IDispatch`. При объявлении их методы обязательно должны вызывать директиву `safecall`. Параметры и возвращаемые результаты методов должны иметь только такие типы: `Byte`, `Currency`, `Real`, `Double`, `Real48`, `Integer`, `Single`, `Smallint`, `AnsiString`, `ShortString`, `TDateTime`, `Variant`, `OleVariant` и `WordBool`.

Такие интерфейсы реализуют связывание, обеспеченное компиляцией приложения (виртуальная таблица), и связывание во время выполнения или позднее связывание (метод `Invoke`).

Первые три метода дуального интерфейса наследуются от `IUnknown`, следующие четыре — от `IDispatch`, затем идут собственные методы интерфейса.

Объект Автоматизации

Возможности объекта Автоматизации в Delphi инкапсулирует класс `TAutoObject`. Он наследует от классов `TCOMObject` и `TTypedCOMObject` (рис. 25.2).

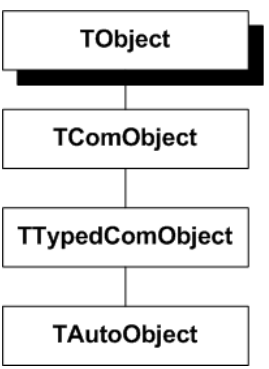


Рис. 25.2. Иерархия предков класса `TAutoObject`

Сначала рассмотрим его основные свойства и методы, а затем приведем пример создания объекта Автоматизации.

Класс *TAutoObject*

Так как класс объекта Автоматизации `TAutoObject` инкапсулирует интерфейс `IDispatch`, он имеет четыре метода, аналогичные методам интерфейса. Это методы `Invoke`, `GetTypeInfo`, `GetTypeInfoCount` и `GetIDsOfNames`. И конечно, в качестве наследника класса `TComObject` класс объекта Автоматизации наследует и три метода интерфейса `IUnknown`.

Для создания экземпляра объекта Автоматизации, так же, как и для любого объекта COM, необходима фабрика класса. В созданном объекте на выбранную фабрику класса указывает свойство

```
property AutoFactory: TAutoObjectFactory;
```

При создании объекта вызывается метод

```
procedure Initialize; override;
```

Для класса `TAutoObject` он пуст, однако разработчики могут перекрыть его в классах-потомках и реализовать там необходимые при инициализации объекта действия.

Для создания нового объекта Автоматизации используйте Репозиторий Delphi. При щелчке на значке **Automation Object** в категории **Delphi Projects | ActiveX** появляется окно мастера создания объекта Автоматизации (рис. 25.3).

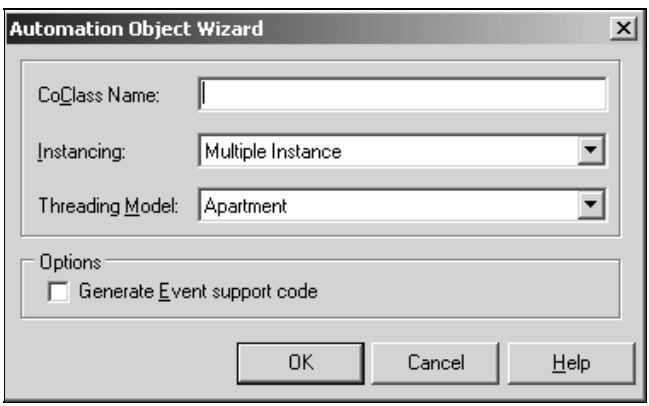


Рис. 25.3. Мастер создания объекта Автоматизации

Поле **CoClass Name** служит для задания имени нового объекта.

Список **Instancing** определяет способ запуска сервера. Список **Threading Model** определяет способ взаимодействия объекта и контроллеров. Возможные значения этих параметров подробно описаны в *главе 24*.

Флажок **Generate Event support code** добавляет к коду создаваемого объекта Автоматизации дополнительный интерфейс, который обеспечивает управление событиями объекта на стороне сервера.

Одновременно с объектом автоматизации создается одноименный `CoClass` и два интерфейса: одноименный объекту интерфейс, наследующий от интерфейса `IDispatch`, а также диспинтерфейс.

Как и для любого другого объекта COM, `CoClass` объекта Автоматизации имеет два метода класса `Create` и `CreateRemote` для создания экземпляра фабрики класса объекта.

Например, при создании объекта Автоматизации `SomeAuto` в новом модуле будет сгенерирован следующий тип:

```
type
  TSomeAuto = class(TAutoObject, ISomeAuto)
  protected
    { Protected declarations }
  end;
```

который наследует от класса `TAutoObject` и имплементирует методы одноименного интерфейса, созданного для поддержки нового объекта.

По аналогии с объектами COM, в секции инициализации модуля нового объекта создается экземпляр фабрики класса:

```
initialization
  TAutoObjectFactory.Create(ComServer, TSomeAuto, Class_SomeAuto,
    ciMultiInstance, tmApartment);
```

Новый интерфейс объявлен в Библиотеке типов (если она уже существует в проекте, то новые сущности добавляются в Библиотеку, иначе она создается):

```
  ISomeAuto = interface(IDispatch)
    ['{9C2A1E76-391E-11D6-A80B-BA576D9FB37E}']
  end;


  ISomeAutoDisp = dispinterface
    ['{9C2A1E76-391E-11D6-A80B-BA576D9FB37E}']
  end;
```

Обратите внимание, что для основного интерфейса создан и диспинтерфейс. И в дальнейшем при добавлении методов в основном интерфейсе эти же методы будут добавлены и в диспинтерфейс и им будут присвоены идентификаторы диспетчеризации.

Здесь же, в Библиотеке типов, объявлен и `CoClass` для нового объекта Автоматизации.

Обработка событий объекта Автоматизации

Если при создании объекта Автоматизации в мастере включить флажок **Generate Event support code**, то к новому объекту будет добавлен механизм обработки событий, который обеспечивает передачу контроллеру Автоматизации событий, возникающих в объекте Автоматизации.

Для того чтобы сервер мог обращаться к контроллеру, объект Автоматизации (и любой другой объект COM) должен поддерживать хотя бы один *исходящий интерфейс.*

Свойство

```
property EventSink: IUnknown;
```

как раз и предоставляет такой интерфейс. Он обеспечивает передачу контроллеру событий, происходящих в объекте.

Объект с исходящими интерфейсами называется *объектом с подключением* и должен поддерживать интерфейс `IConnectionPointContainer`, через который клиент может узнать о доступных исходящих интерфейсах.

Каждому исходящему интерфейсу должен соответствовать специальный объект, реализующий его методы. Такой объект называется *приемником* (sink).

Рассмотрим исходный код (листинг 25.1), который автоматически генерируется при включении флажка **Generate Event support code**.

Листинг 25.1. Библиотека типов объекта Автоматизации (комментарии исключены)

```
unit Project1_TLB;
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
const
  LIBID_Project1: TGUID = '{B2211452-E7A8-11D2-80F3-008048A9D587}';
  IID_ISomeObj: TGUID = '{B2211453-E7A8-11D2-80F3-008048A9D587}';
  DIID_ISomeObjEvents: TGUID = '{B2211455-E7A8-11D2-80F3-008048A9D587}';
  CLASS_SomeObj: TGUID = '{B2211457-E7A8-11D2-80F3-008048A9D587}';
type
  ISomeObj = interface;
  ISomeObjDisp = dispinterface;
  ISomeObjEvents = dispinterface;
SomeObj = ISomeObj;
ISomeObj = interface(IDispatch)
```

```
    ['{B2211453-E7A8-11D2-80F3-008048A9D587}']
  end;
ISomeObjDisp = dispinterface
    ['{B2211453-E7A8-11D2-80F3-008048A9D587}']
  end;
ISomeObjEvents = dispinterface
    ['{B2211455-E7A8-11D2-80F3-008048A9D587}']
  end;
  CoSomeObj = class
    class function Create: ISomeObj;
    class function CreateRemote(const MachineName: string): ISomeObj;
  end;
implementation
uses ComObj;
class function CoSomeObj.Create: ISomeObj;
begin
  Result := CreateComObject(CLASS_SomeObj) as ISomeObj;
end;
class function CoSomeObj.CreateRemote(const MachineName: string):
ISomeObj;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_SomeObj) as
ISomeObj;
end;
end.
```

Как видно из листинга, для объекта Автоматизации создаются три интерфейса. Дуальный интерфейс `ISomeObj` предназначен для работы с объектом `TSomeObj`. Интерфейс диспетчеризации `ISomeObjDisp` обеспечивает работу Автоматизации. Обработку событий объекта в контроллере обеспечивает интерфейс `ISomeObjEvents`. Листинг 25.2 содержит код модуля объекта Автоматизации.

Листинг 25.2. Модуль объекта Автоматизации

```
unit Unit2;
interface
uses ComObj, ActiveX, AxCtrls, Project1_TLB;
type
  TSomeObj = class(TAutoObject, IConnectionPointContainer, ISomeObj)
```

```
  private
    FConnectionPoints: TConnectionPoints;
    FEvents: ISomeObjEvents;
  public
    procedure Initialize; override;
  protected
      property ConnectionPoints: TConnectionPoints read FConnectionPoints
      implements IConnectionPointContainer;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
  end;
implementation
uses ComServ;
procedure TSomeObj.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as ISomeObjEvents;
end;
procedure TSomeObj.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
      ckSingle, EventConnect);
end;

initialization
  TAutoObjectFactory.Create(ComServer, TSomeObj, Class_SomeObj,
    ciMultiInstance, tmApartment);
end.
```

В самом объекте интерфейс `ISomeObjEvents` передается в поле `FEvents`. Для его инициализации автоматически перекрывается метод `EventSinkChanged`. Интерфейс `ISomeObjEvents` основывается на возможностях интерфейса `IConnectionPointContainer`, который включен в объявление класса.

Обратите внимание, что инициализация дополнительных объектов осуществляется перекрытым методом `Initialize`.

Программный код, представленный в листингах 25.1 и 25.2, создается автоматически.

Если разработчик хочет передавать событие в контроллер, то он должен обеспечить выполнение следующих действий:

- для каждого обрабатываемого события должен быть создан метод интерфейса обработки событий;
- для интерфейса обработки событий создается `CoClass`;
- при инициализации объекта Автоматизации необходимо создавать экземпляр класса обработки событий.

Рассмотрим все перечисленные этапы более подробно с примерами исходного кода для объекта Автоматизации `TSomeObject`.

Создание методов-аналогов событий

Интерфейс обработки событий должен иметь методы, соответствующие обрабатываемым событиям:

```
ISomeObjEvents = dispinterface
    ['{B2211455-E7A8-11D2-80F3-008048A9D587}']
    procedure Open; dispid 1;
    procedure Close; dispid 2;
  end;
```

Создание класса-оболочки

Для интерфейса обработки событий необходимо вручную создать класс-оболочку. Этот класс должен содержать выбранные события и предусматривать их обработку в методе `Invoke` (листинг 25.3).

Листинг 25.3. Реализация класса-оболочки

```
TSomeEventSink = class(TInterfacedObject, IUnknown, IDispatch)
  private
   {...}
   FOwner : TObject;
   FOnOpen : TNotifyEvent;
   FOnClose : TNotifyEvent;
   {...}
  protected
   function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
  public
   {...}
   property OnOpen: TNotifyEvent read FOnOpen write FOnOpen;
```

```
   property OnClose: TNotifyEvent read FOnClose write FOnClose;
  end;
  ...
function TSomeEventSink.Invoke(DispID: Integer; const IID: TGUID;
LocaleID: Integer; Flags: Word; var Params; VarResult, ExcepInfo, ArgErr:
Pointer): HRESULT;
begin
 ...
 case dispid of
  5: if Assigned(FOnOpen) then FOnOpen(FOwner);
  6: if Assigned(FOnClose) then FOnClose(FOwner);
 end;
 ...
end;
```

Инициализация объекта Автоматизации

Объект Автоматизации при инициализации должен создавать экземпляр класса `TSomeEventSink` и уже в собственных событиях вызывать аналогичные события класса обработки событий (листинг 25.4).

Листинг 25.4. Инициализация объекта Автоматизации

```
TSomeObject = class
  private
   {...}
   FEventSink: TSomeEventSink;
   function GetOnOpen: TNotifyEvent;
   procedure SetOnOpen(Value: TNotifyEvent);
   function GetOnClose: TNotifyEvent;
   procedure SetOnClose(Value: TNotifyEvent);
  public
   constructor Create;
   {...}
published
   {...}
   property OnOpen: TNotifyEvent read GetOnOpen write SetOnOpen;
   property OnClose: TNotifyEvent read GetOnClose write SetOnClose;
end;
```

```
{...}
function TSomeObj.GetOnOpen: TNotifyEvent;
begin
  Result := FEventSink.OnOpen;
end;
procedure TSomeObj.SetOnOpen(Value: TNotifyEvent);
begin
  FEventSink.OnOpen := Value;
end;
function TSomeObj.GetOnClose: TNotifyEvent;
begin
  Result := FEventSink.OnClose;
end;
procedure TSomeObj.SetOnClose(Value: TNotifyEvent);
begin
  FEventSink.OnClose := Value;
end;
```

Фабрика класса

Фабрика класса для объекта Автоматизации создается на основе класса `TAutoObjectFactory`. Фабрика класса Автоматизации наследует от стандартной фабрики класса COM (рис. 25.4).

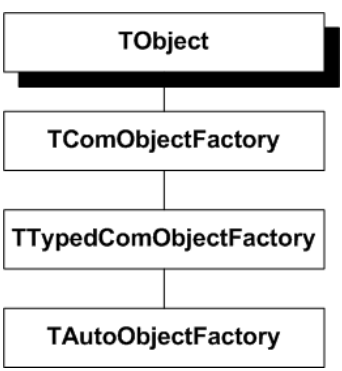


Рис. 25.4. Иерархия классов-предков фабрики класса Автоматизации

Фабрика класса Автоматизации создается конструктором

```
constructor Create(ComServer: TComServerObject; AutoClass: TAutoClass;
const ClassID: TGUID; Instancing: TClassInstancing;ThreadingModel:
TThreadingModel = tmSingle);
```

Как видно из объявления, назначения его параметров совпадают с параметрами конструктора фабрики класса COM *(см. главу 24)*.

После создания фабрика класса обеспечивает доступ к информации об интерфейсе диспетчеризации объекта.

Свойство

```
PInterfaceEntry = ^TInterfaceEntry;
  TInterfaceEntry = packed record
    IID: TGUID;
    VTable: Pointer;
    IOffset: Integer;
    ImplGetter: Integer;
  end;
property DispIntfEntry: PInterfaceEntry;
```

возвращает ссылку на соответствующую структуру.

Метод

```
function GetIntfEntry(Guid: TGUID): PInterfaceEntry; virtual;
```

возвратит информацию об диспинтерфейсе, заданном параметром `GUID`.

Информация о типе диспинтерфейса созданного фабрикой объекта Автоматизации возвращается свойством

```
property DispTypeInfo: ITypeInfo;
```

Поддержку рассмотренного интерфейса обработки событий обеспечивает свойство

```
property EventIID: TGUID;
```

возвращающее `GUID` интерфейса обработки событий. Информацию о типе для этого интерфейса содержит свойство

```
property EventTypeInfo: ITypeInfo;
```

Класс *TAutoIntfObject*

Если объект Автоматизации необходимо использовать внутри приложения, то можно применить класс `TAutoIntfObject`, который для создания экземпляра класса не требует фабрики класса. Он поддерживает интерфей-

сы `IDispatch` и `ISupportErrorInfo` и выполняет ряд функций, присущих фабрике класса.

Его конструктор

```
constructor Create(const TypeLib: ITypeLib; const DispIntf: TGUID);
```

создает объект Автоматизации с глобальным идентификатором `DispIntf` на основе информации о типе `TypeLib`.

После создания диспинтерфейс объекта Автоматизации доступен через свойство

```
property DispIID: TGUID;
```

Информацию о типе диспинтерфейса обеспечивает свойство

```
property DispTypeInfo: ITypeInfo;
```

А структура `TInterfaceEntry` (см. выше) доступна через свойство

```
property DispIntfEntry: PInterfaceEntry;
```

Сервер Автоматизации

Как создать в Delphi сервер Автоматизации? Любое приложение, имеющее в своем составе нормально функционирующий объект Автоматизации и зарегистрированное в операционной системе соответствующим образом, становится сервером Автоматизации.

В соответствии с канонами COM, можно создавать три типа серверов Автоматизации:

- ❒ внутренние (in-process);
- ❒ внешние (out-process);
- ❒ удаленные (remote).

Объект автоматизации инкапсулирован в классе `TAutoObject`, который является потомком основного класса COM — `TCOMObject`. Для включения объекта Автоматизации в приложение существует мастер, открывающийся при щелчке на значке **Automation Object** в категории **Delphi Projects | ActiveX** в Репозитории Delphi. Об этом рассказывается далее.

Примечание

Напомним, что объекты COM, а значит, и объекты Автоматизации тоже, содержат программный код для методов связанных с ними интерфейсов.

Регистрация приложения в операционной системе в качестве внутреннего сервера автоматизации выполняется командой **Register ActiveX Server** из ме-

ню **Run**. Команда **Unregister ActiveX Server** применяется для отмены регистрации.

Регистрация внешнего сервера осуществляется ключом `/regserver` при запуске приложения. Для отмены регистрации применяется ключ `/unregserver`.

После регистрации к серверу может обратиться любое приложение, имеющее информацию о методах его интерфейса.

Контроллер автоматизации

Контроллер автоматизации управляет работой сервера, используя методы принадлежащих серверу интерфейсов. Информация об интерфейсах обычно распространяется в виде библиотек типов (подробнее о языках библиотек типов *см. главу 24*). Любое приложение, имеющее доступ к соответствующей библиотеке типов, может стать контроллером автоматизации.

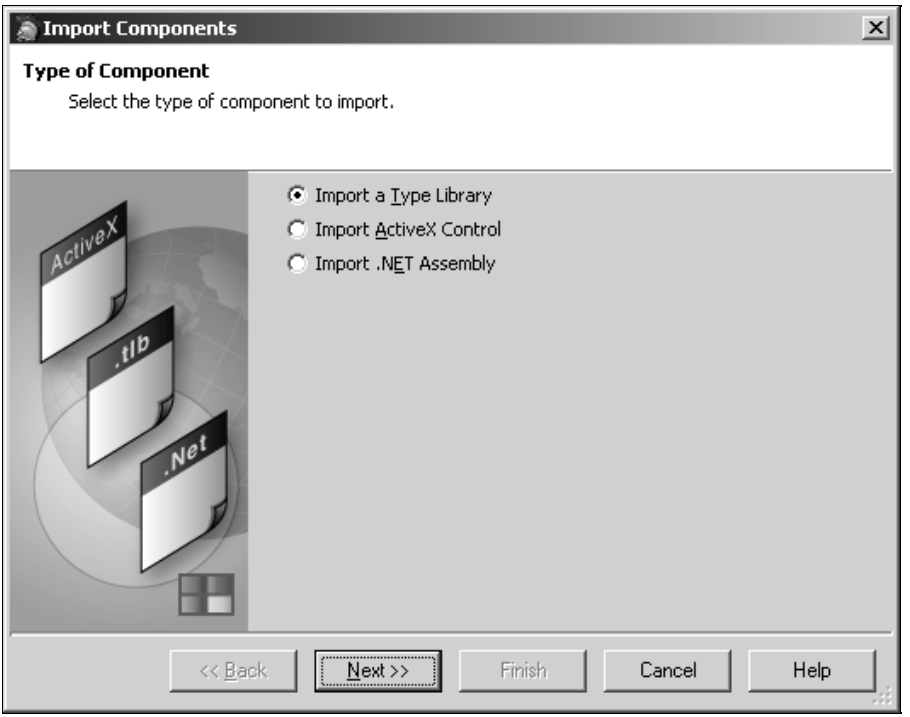


Рис. 25.5. Первая страница мастера импорта библиотеки типов

Для включения библиотеки типов в состав проекта служит команда **Import Component** из меню **Component**, которая открывает окно специализирован-

ного мастера. Или же, если библиотека создана в Delphi, можно просто включить имя ее файла в секцию `uses` модуля.

На первой странице мастера необходимо выбрать переключатель **Import a Type Library** (рис. 25.5).

На следующей странице требуется из списка выбрать нужную библиотеку типов (рис. 25.6). В этом списке содержатся все зарегистрированные в системе серверы Автоматизации. Для любого из них можно импортировать библиотеку типов.

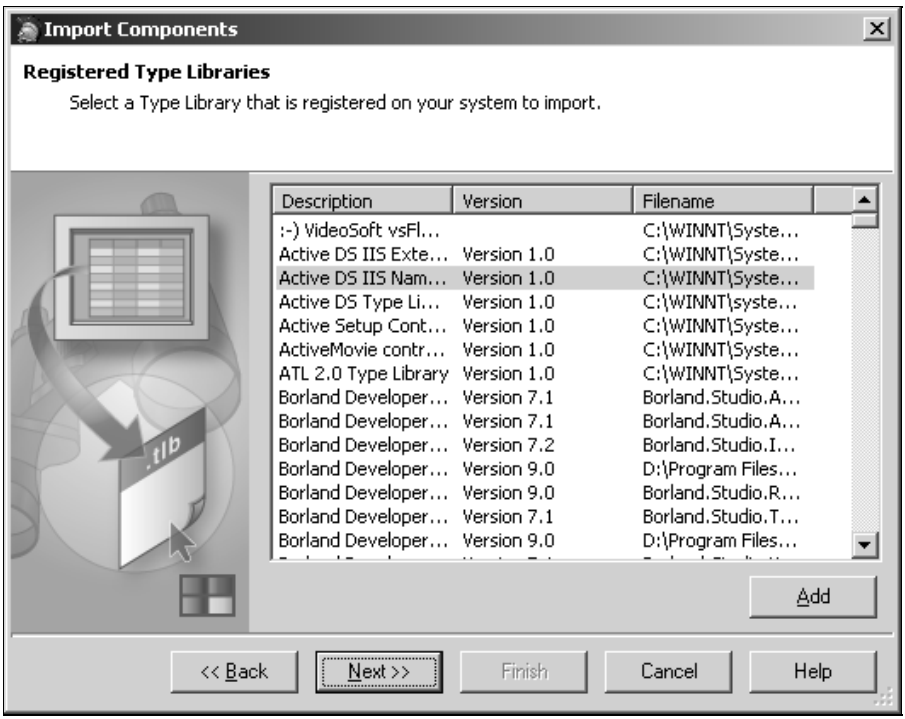


Рис. 25.6. Выбор зарегистрированного сервера Автоматизации в мастере импорта библиотеки типов

На следующей странице указываются параметры, представленные на рис. 25.7. В поле **Class Name(s)** перечисляются классы библиотеки типов выбранного сервера Автоматизации. В поле **Unit dir Name** указывается путь к файлу сервера Автоматизации, содержащему библиотеку типов.

Затем требуется указать в качестве приемника библиотеки типов модуль (рис. 25.8).

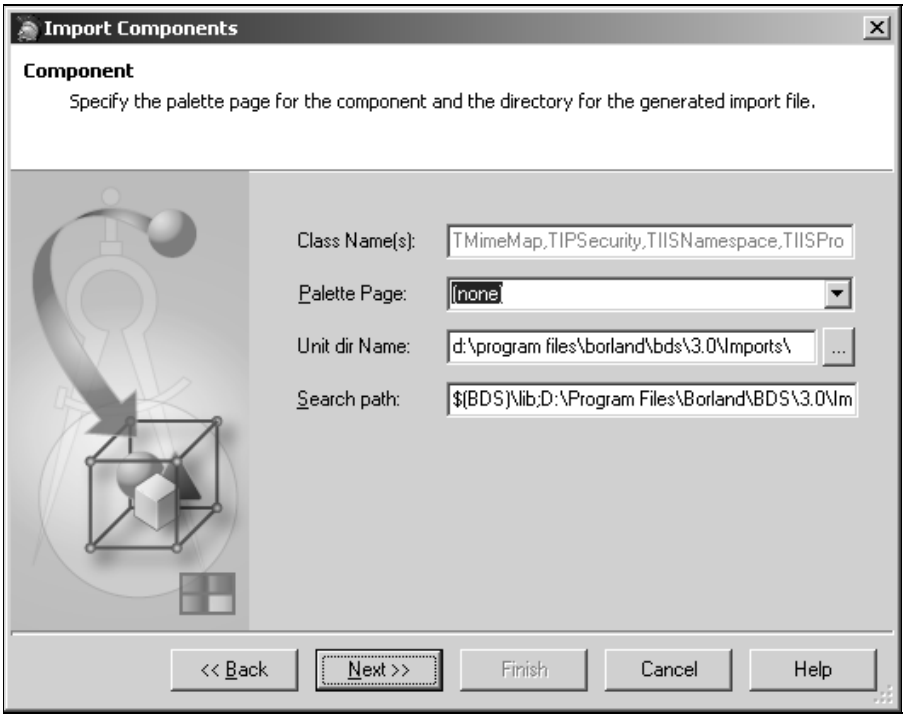


Рис. 25.7. Страница параметров мастера импорта библиотеки типов

После этого в контроллере необходимо создать экземпляр класса-оболочки интерфейса (`CoClass`) и применить в нужных местах требуемые методы сервера Автоматизации.

Если библиотека типов недоступна, разработчик должен поступить следующим образом.

1. Сначала на основе переменной типа `OleValiant` в приложении-контроллере создается объект Автоматизации. Для этого предусмотрена функция

   ```
   function CreateOleObject(const ClassName: string): IDispatch;
   ```

   которая возвращает указатель на интерфейс `IDispatch`. Параметр `ClassName` определяет имя класса Автоматизации. Это идентификатор программы, являющейся сервером Автоматизации, он соответствует идентификатору класса `CLSID`. Эта функция пригодна для внутренних и внешних серверов. Естественно, сервер, для которого создается объект, должен быть зарегистрирован.

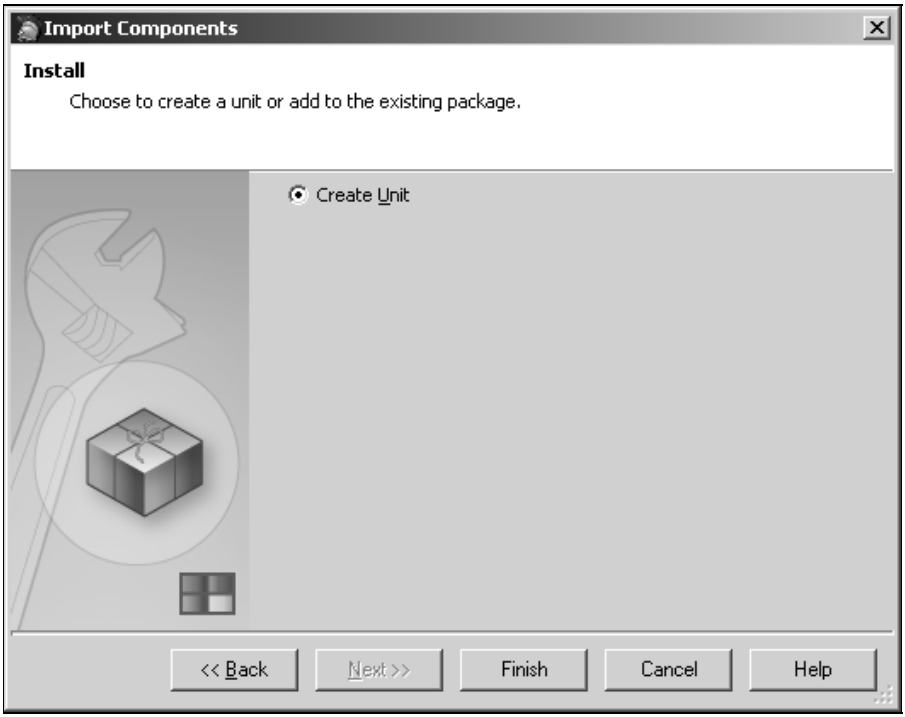


Рис. 25.8. Выбор приемника библиотеки типов мастера импорта

2. После создания объекта Автоматизации методы его интерфейсов доступны в приложении-контроллере. Если при первом обращении сервер не работает, то он автоматически запускается операционной системой.

Пример приложения Автоматизации

Теперь рассмотрим простейший пример создания сервера и контроллера Автоматизации. В качестве модели выберем внешний сервер. Так как нам важно показать принцип работы механизма Автоматизации в приложениях Delphi, ограничим сервер выполнением элементарнейших функций. Впоследствии работающий простейший сервер вы сможете самостоятельно дополнить реальными операциями.

Сервер Автоматизации

Создание сервера начинается с открытия в Delphi проекта обычного приложения. Разработчик может запрограммировать для него любые необходимые

операции. Приложение становится сервером Автоматизации после включения в его состав объекта Автоматизации.

Для этого удобен Репозиторий объектов (рис. 25.9). В категории **Delphi Projects | ActiveX** необходимо выбрать значок **Automation Object**.

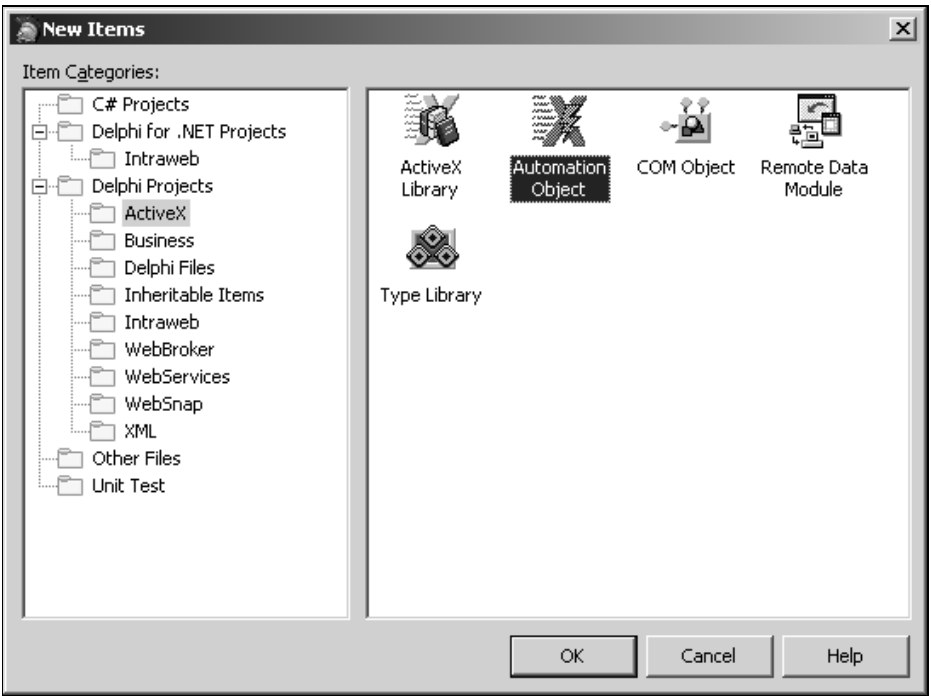


Рис. 25.9. Репозиторий Delphi при создании объекта Автоматизации

При создании нового объекта Автоматизации появляется диалог **Automation Object Wizard** (см. рис. 25.3). Подробно процесс создания объекта рассмотрен ранее в этой главе. Полностью в соответствии с этим описанием был создан объект `TSimple` (табл. 25.1).

Таблица 25.1. *Параметры объекта* `TSimple`

Параметр	**Значение**
`CoClass Name`	`Simple`
`Instancing`	`tmApartment`
`Threading Model`	`ciMultiInstance`
`Generate Event support code`	Включен

Создание и настройка свойств и методов интерфейсов объекта осуществляются в Редакторе библиотеки типов. В нем были созданы объявления для двух методов: `Method1` и `Method2`. Подробнее о создании методов интерфейсов *см. главу 24*.

Обратите внимание (листинг 25.5), что интерфейс `ISimple` объекта Автоматизации полностью соответствует требованиям к дуальному интерфейсу. Наряду с ним автоматически создается интерфейс диспетчеризации `ISimpleDisp`. В остальном исходный код библиотеки типов соответствует стандартному объекту COM.

Листинг 25.5. Библиотека типов объекта Автоматизации `TSimple` (комментарии удалены)

```
unit DemoAutoServer_TLB;
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
const
  LIBID_DemoAutoServer: TGUID = '{F290AEE0-E858-11D2-80F3-008048A9D587}';
  IID_ISimple: TGUID = '{F290AEE1-E858-11D2-80F3-008048A9D587}';
  CLASS_Simple: TGUID = '{F290AEE3-E858-11D2-80F3-008048A9D587}';
type
ISimple = interface;
  ISimpleDisp = dispinterface;
Simple = ISimple;
ISimple = interface(IDispatch)
    ['{F290AEE1-E858-11D2-80F3-008048A9D587}']
    procedure Method1; safecall;
    procedure Method2; safecall;
  end;
ISimpleDisp = dispinterface
    ['{F290AEE1-E858-11D2-80F3-008048A9D587}']
    procedure Method1; dispid 1;
    procedure Method2; dispid 2;
  end;
  CoSimple = class
    class function Create: ISimple;
    class function CreateRemote(const MachineName: string): ISimple;
  end;
implementation
uses ComObj;
```

```
class function CoSimple.Create: ISimple;
begin
  Result := CreateComObject(CLASS_Simple) as ISimple;
end;
class function CoSimple.CreateRemote(const MachineName: string): ISimple;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_Simple) as ISimple;
end;
end.
```

Как и обычному интерфейсу COM, дуальному интерфейсу автоматизации должен соответствовать специальный объект-оболочка, в котором реализованы свойства и методы интерфейса (`CoClass`). Листинг 25.6 содержит код модуля объекта Автоматизации `TSimple`.

Листинг 25.6. Модуль объекта Автоматизации `TSimple`

```
unit SimpleObj;
interface
uses
  ComObj, ActiveX, Dialogs, DemoAutoServer_TLB;
type
  TSimple = class(TAutoObject, ISimple)
  protected
    procedure Method1; safecall;
    procedure Method2; safecall;
  end;
implementation
uses ComServ;
procedure TSimple.Method1;
begin
 ShowMessage('Method2 executed');
end;

procedure TSimple.Method2;
begin
 ShowMessage(' Method2 executed');
end;
initialization
```

```
  TAutoObjectFactory.Create(ComServer, TSimple, Class_Simple,
    ciMultiInstance, tmApartment);
end.
```

После завершения разработки приложения-сервера необходимо зарегистрировать его в операционной системе. Для этого достаточно запустить приложение с ключом `/regserver`. Его можно поместить в диалоге параметров запуска (команда **Parameters** меню **Run**).

Контроллер Автоматизации

Приложение-контроллер Автоматизации должно иметь информацию о возможностях интерфейсов, предоставляемых сервером Автоматизации. Для этого оно должно импортировать библиотеку типов или включить соответствующие модули в секцию `uses`. Второй способ хорош в случае, если сервер и контроллер разрабатываются одновременно. Он применен в рассматриваемом нами примере, т. к. сервер и контроллер расположены в одной группе проектов.

Форма контроллера (листинг 25.7) включает лишь две кнопки, которые будут использованы для вызова методов сервера Автоматизации.

Листинг 25.7. Модуль главной формы контроллера Автоматизации

```
unit ContrMain;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Buttons, DemoAutoServer_TLB;
type
  TContrForm = class(TForm)
    Label1: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    procedure FormCreate(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
   public
    Simple: ISimple;
  end;
```

```
var ContrForm: TContrForm;
implementation
{$R *.DFM}
procedure TContrForm.FormCreate(Sender: TObject);
begin
 Simple := CoSimple.Create;
end;
procedure TContrForm.BitBtn1Click(Sender: TObject);
begin
 Simple.Method1;
end;
procedure TContrForm.BitBtn2Click(Sender: TObject);
begin
 Simple.Method2;
end;
end.
```

При создании формы контроллера в методе `FormCreate` создается экземпляр объекта `CoClass` для объекта Автоматизации. Конструктор возвращает указатель на интерфейс `ISimple`.

Щелчок на кнопках формы контроллера Автоматизации вызывает соответствующие методы интерфейса `ISimple` сервера `DemoAutoServer`. Если при этом сервер не работал, операционная система запускает его автоматически.

Резюме

Автоматизация основана на технологии COM. Основой Автоматизации является интерфейс `IDispatch`.

Сервер Автоматизации предоставляет клиентам свои интерфейсы и должен быть зарегистрирован в операционной системе. Контроллер Автоматизации использует возможности сервера.

Для организации взаимодействия между сервером и контроллером интерфейс должен соответствовать ряду условий. Это может быть интерфейс диспетчеризации или дуальный интерфейс.

Глава 26

Элементы управления ActiveX

В предыдущих главах рассматривались вопросы создания объектов COM и Автоматизации, которые предоставляют клиентам наборы стандартных (определенных в рамках соответствующей спецификации) и собственных интерфейсов. Методы и свойства этих интерфейсов доступны другим приложениям. Тем самым реализуется программный интерфейс COM для таких объектов.

Теперь настало время расширить область применения объектов COM на визуальный пользовательский интерфейс приложения. В самом деле, если объект предоставляет для использования свои методы, почему бы ему не сделать то же самое и для своей визуальной функциональности?

Конечно же, визуальное представление объектов COM еще нужно реализовать. Эту задачу как раз и выполняет еще одна дочерняя технология COM — *технология ActiveX*.

В рамках технологии ActiveX реализованы процессы разработки и распространения элементов управления на основе объектов COM — *элементов управления ActiveX*. Такие элементы управления могут обеспечивать самые разнообразные дополнительные функции визуального интерфейса приложений, предоставляя пользователям дополнительные удобства и возможности.

Компонентное программирование начиналось с Microsoft Visual Basic. Именно в этой среде впервые были применены 16-разрядные модули расширения VBX (Visual Basic eXtension). Наличие такого механизма очень заинтересовало разработчиков, и в течение короткого срока возможности этой среды программирования значительно углубились и расширились — за счет появления сотен и тысяч новых VBX.

Идея компонентной разработки получила развитие, и скоро по этому принципу стали строиться и другие продукты (например, Delphi). С приходом 32-разрядного программирования компонентный подход еще более развился, и теперь для него есть единый стандарт — ActiveX.

Параллельно с этими процессами активно развивались технологии Internet и WWW. И оказалось, что первые элементы управления на основе объектов

COM плохо подходят для этих новых областей применения. Суть проблемы заключалась в том, что эти элементы управления (тогда они еще назывались элементы управления OLE) в соответствии со спецификацией обязаны были поддерживать слишком много стандартных интерфейсов. И это при том, что в большинстве элементов управления они никогда не использовались. В результате элементы управления получались громоздкими и долго загружались средствами Internet-технологий.

Требования были существенно облегчены, элементы управления стали гораздо более компактными и дали толчок к развитию ряда смежных технологий. И за ними было закреплено название "элементы управления ActiveX".

С точки зрения программиста, ActiveX — это черный ящик, обладающий свойствами, методами и событиями. ActiveX встроены в Delphi настолько органично, что вы можете ничего не знать о COM, и тем не менее задействовать всю полноту их возможностей. С точки зрения модели объектов COM, элемент управления ActiveX — это сервер, поддерживающий технологию Автоматизации, реализованный в виде динамической библиотеки, исполняемый в адресном пространстве вашего приложения и допускающий визуальное редактирование.

В этой главе мы рассмотрим следующие вопросы.

- Как работают элементы управления ActiveX.
- Как включить в состав Delphi и применить готовые элементы управления, созданные сторонними разработчиками.
- Как стандартные компоненты Delphi превратить в компоненты ActiveX.
- Как преобразовать форму в форму ActiveX.

Как работают элементы управления ActiveX

Элемент управления ActiveX является объектом COM, функционирующим чаще всего в составе внутреннего сервера *(см. главу 24)*. Конечно же, технология ActiveX накладывает на элементы управления ряд дополнительных требований, но все они обусловлены необходимыми механизмами загрузки и функционирования элементов управления.

Являясь частью визуального интерфейса, элементы управления должны взаимодействовать с приложением, передавая ему адекватную информацию о действиях пользователя, реагировать самостоятельно на эти действия, корректно изменять собственное состояние в соответствии с сигналами приложения (рис. 26.1).

Например, если пользователь нажал кнопку элемента управления ActiveX, то этот элемент должен изобразить нажатие кнопки и известить об этом приложение. А если, к примеру, приложение посчитает необходимым сде-

лать свои элементы управления недоступными, то и наш элемент должен суметь сделать это, реагируя на сигнал приложения.

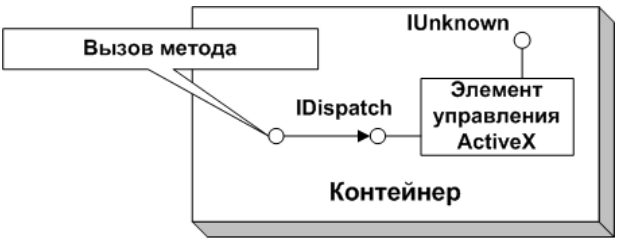


Рис. 26.1. Для доступа к элементу управления ActiveX используются диспинтерфейсы

С формальной точки зрения элемент управления ActiveX обязан поддерживать лишь интерфейс `IUnknown`. Однако для обеспечения основных функций, по определению присущих элементам управления ActiveX, он должен реализовывать некоторое подмножество из набора базовых интерфейсов ActiveX. Их перечень представлен в табл. 26.1.

Таблица 26.1. *Базовые интерфейсы элементов управления ActiveX*

Интерфейс	Описание
`IOleObject`	Предоставляет важнейшие методы для взаимодействия элемента управления с контейнером
`IOleControl`	Поддерживает базовую функциональность элемента управления: установку свойств и реакцию на события
`IOleInPlaceObject`	Интерфейс управляет активизацией и деактивизацией элемента управления в контейнере
`IOleInPlaceActiveObject`	Обеспечивает прямой канал для взаимодействия между элементом управления и окном-рамкой контейнера, содержащим данный элемент
`IObjectSafety`	Интерфейс предоставляет два метода для установки режима безопасного использования элемента управления. Если установить этот режим, то например, при загрузке через Internet-браузер будет получать информацию, что данный элемент безопасен для окружения, в котором он загружается, и для используемых данных
`IPersistPropertyBag`	Предоставляет методы для сохранения и загрузки индивидуальных свойств элемента управления в хранилище

Таблица 26.1 *(окончание)*

Интерфейс	Описание
`IPersistStreamInit`	Обеспечивает инициализацию хранилища, используемого для сохранения значений свойств элемента управления на основе использования потоков внутри хранилища
`IPersistStorage`	Обеспечивает для элемента управления ссылку на экземпляр хранилища, предоставленного контейнером для сохранения свойств элемента
`IPerPropertyBrowsing`	Предоставляет методы для передачи значений свойств элемента управления в страницу свойств
`ISimpleFrameSite`	Обеспечивает функционирование элемента управления как оконного элемента для размещения других элементов управления AcitveX
`ISpecifyPropertyPages`	Содержит метод, возвращающий информацию о наличии и поддержке страниц свойств данным элементом управления
`IViewObject`	Обеспечивает взаимодействие элемента управления и контейнера при необходимости сообщить контейнеру об изменениях во внешнем виде элемента управления
`IViewObject2`	Расширение интерфейса `IViewObject`. Предоставляет информацию о размерах области отрисовки элемента управления

Доступ к методам интерфейсов осуществляется через метод `Invoke` интерфейса `IDispatch`.

Итак, для элементов управления ActiveX можно выделить следующие основные функции:

- регистрация элемента управления;
- реализация пользовательского интерфейса и визуализация;
- предоставление своих методов и свойств приложению;
- нотификация приложения о возникающих событиях;
- получение информации о свойствах приложения.

Технология ActiveX как раз определяет и реализует стандарты поведения компонентов пользовательского интерфейса приложения.

Давайте посмотрим, как решаются эти задачи.

Контейнеры и регистрация элементов управления ActiveX

Перечисленные задачи решаются за счет того, что любые элементы управления AcitveX предоставляют стандартные интерфейсы. Эти интерфейсы способны поддерживать объекты COM в заданном состоянии и эффективно взаимодействовать с приложением, использующим элемент.

Приложение, имеющее в своем составе компоненты ActiveX и умеющее их задействовать, называется *контейнером элементов управления*. Пользователи работают со многими программами и даже не подозревают, что они являются полноценными контейнерами ActiveX. К ним, например, можно отнести Microsoft Internet Explorer, Microsoft Visual Basic, Microsoft Word. Ну и, конечно, Delphi также является контейнером.

Подобно тому, как элементы управления ActiveX не обязаны поддерживать все стандартные интерфейсы, контейнеры также не должны обеспечивать возможность применения любого компонента ActiveX. Если приложение способно реализовать лишь малую часть возможностей компонента ActiveX, например, его визуализацию и реакцию на действия пользователя, тем не менее оно является контейнером. Перечень базовых интерфейсов для контейнеров ActiveX представлен в табл. 26.2.

Таблица 26.2. *Базовые интерфейсы элементов управления ActiveX*

Интерфейс	Описание
`IAdviseSink`	Дает возможность контейнеру отслеживать изменения в элементах управления, связанных документах и т. д.
`IOleClientSite`	При помощи этого интерфейса присоединенные объекты могут получать информацию о ресурсах контейнера
`IOleDocumentSite`	Применяется для активизации документов OLE
`IOleInPlaceSite`	Управляет взаимодействием между контейнером и сайтом своего элемента управления
`IOleUIObjInfo`	Предоставляет информацию об элементах управления контейнера для использования в диалогах настройки свойств контейнера

Свои функции контейнеры также реализуют, предоставляя методы стандартных интерфейсов.

Для того чтобы элемент управления ActiveX мог быть использован приложением, он должен быть зарегистрирован на данном компьютере. Для этого при установке элементы управления ActiveX заносят сведения о себе в системный реестр. Подробнее процедура регистрации рассматривается далее.

Предоставление методов

Все стандартные интерфейсы контейнеров и компонентов ActiveX являются интерфейсами диспетчеризации и происходят от `IDispatch`. Соответственно любые вызовы методов элементов управления из контейнера осуществляются через метод `Invoke` интерфейса `IDispatch` *(см. главу 25)*. На самом деле контейнеру необходим только интерфейс `Idispatch`, и ничего кроме него.

Для получения информации о методах диспинтерфейсов контейнер обращается к библиотеке типов, предоставляемой элементом управления.

События

Реагируя на действия пользователя, элемент управления ActiveX посылает контейнеру информацию о событиях, предоставляя тому возможность реагировать на них. Тем самым элементы ActiveX выполняют свое основное предназначение.

Как же работает нотификация в ActiveX?

При возникновении события элемент управления просто вызывает необходимый метод контейнера. В этом случае контейнер предоставляет свои интерфейсы, а элемент является клиентом. Чтобы эта простая схема заработала, элемент управления должен иметь ссылку на интерфейс контейнера.

Каждому событию элемент управления должен поставить в соответствие метод интерфейса контейнера. Набор таких методов называется *исходящим интерфейсом*. Любые же обычные интерфейсы называются *входящими*.

Однако в этом случае полный, хорошо написанный контейнер должен иметь методы "на все случаи жизни", для любых типов компонентов. Как вы понимаете, такой экстенсивный способ реализации был бы крайне неэффективен.

Поэтому элементы управления ActiveX обязаны предоставить контейнеру библиотеку типов с описанием интерфейсов — как входящих, так и исходящих. Получив информацию о своих методах, используемых элементом ActiveX в исходящих интерфейсах, контейнер должен динамически создать реализацию метода `IDispatch.Invoke`, поддерживающую эти методы.

Свойства

Так как элементы управления ActiveX задействуют диспинтерфейсы, то им доступны свойства интерфейсов *(см. главу 25)*.

Для свойства могут быть определены два метода: чтения и записи. Эти методы, как и любые другие методы диспинтерфейсов, вызываются через `IDispatch.Invoke`.

Страницы свойств

Для стандартизации визуального представления свойств элементов управления ActiveX в них реализованы *страницы свойств*, которые представляют собой совокупность объектов, умеющих отображать значения свойств элемента управления в виде диалогового окна с многостраничным блокнотом.

Если элемент управления поддерживает интерфейс `ISpecifyPropertyPages`, то при помощи единственного метода `GetPages` этого интерфейса контейнер может узнать о свойствах, предоставляемых элементом. После этого контейнер специально для отображения страницы свойств создает объект-рамку. Затем объект-рамка создает *объекты-вкладки свойств*. Каждый такой объект будет соответствовать одной странице блокнота диалога страниц свойств.

Объект-рамка и объект-вкладка взаимодействуют друг с другом при помощи пары интерфейсов: первый предоставляет интерфейс `IPropertyPageSite`, а второй — `IPropertyPage`.

Если пользователь изменил значение свойства в диалоге страниц свойств, соответствующий объект-вкладка обеспечивает передачу нового значения элементу управления.

Лицензирование

Для элементов управления ActiveX, как и для обычных приложений, остро стоит проблема незаконного использования. Снять остроту этой проблемы и призвано *лицензирование*.

При создании элемента управления для него генерируется лицензионный ключ, который сохраняется в файле с расширением lic. Это статический способ создания ключа.

Если же элемент управления загружается через Internet, то для него создается пакетный файл лицензионного ключа (расширение lpk). Ссылка на такой файл должна быть включена в соответствующую страницу HTML. При открытии страницы браузер загружает пакетный файл, получает ключ и сравнивает его с имеющимся. Такой метод лицензирования называется динамическим.

При создании экземпляра элемента управления контейнер проверяет наличие лицензии на применение этого элемента на данной машине при помощи методов интерфейса `IClassFactory2`.

Метод `GetLicInfo` проверяет наличие глобальной лицензии. Если она имеется, то все в порядке и элемент создается обычным образом. Если нет, контейнер обязан вызвать метод `CreateInstanceLic`, в который в качестве параметра должен передаваться номер лицензии. Фабрика класса элемента

управления проверяет лицензионный ключ, и в случае успеха создает экземпляр компонента.

Для того чтобы получить лицензионный ключ для использования вышеописанного метода, контейнеру нужен еще один метод — `RequestLicKey`, который возвращает искомый ключ (если он, конечно, есть).

Реализация компонентов ActiveX в Delphi

Компоненты ActiveX в Delphi функционируют на основе тех же механизмов, что и обычные объекты COM. Классы компонентов наследуют от класса-предка и от интерфейса-предка. Компоненты ActiveX предоставляют клиентам свои интерфейсы, которые происходят от `IDispatch`. Для создания экземпляров компонентов ActiveX служат фабрики класса. Информация о компоненте содержится в библиотеке типов.

Общим предком всех классов элементов управления ActiveX в Delphi является класс `TActiveXControl`. Он инкапсулирует базовые функции, позволяющие компонентам ActiveX взаимодействовать с контейнером и обеспечивать реакцию компонента на события. Также он содержит все базовые интерфейсы элемента управления ActiveX.

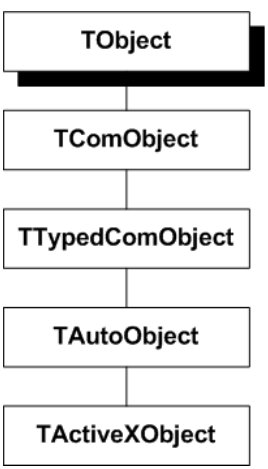


Рис. 26.2. Иерархия классов компонентов ActiveX в Delphi

Из рис. 26.2 очевидно, что технология ActiveX базируется на функциональности своих родительских технологий — COM и Автоматизации.

Процесс создания компонентов ActiveX в Delphi существенно облегчен за счет мастера, генерирующего весь необходимый исходный код для нового компонента, его фабрики класса и библиотеки типов.

Дополнительно к этому для компонентов ActiveX создается специальный класс-надстройка, призванный обеспечить взаимодействие компонента ActiveX со своим контейнером — Delphi. Классы-надстройки порождаются от класса `TOleControl`.

Класс компонента ActiveX

Итак, непосредственным предком любого компонента ActiveX в Delphi является класс `TActiveXControl`. Происходя от базовых классов объектов COM и Автоматизации, он инкапсулирует для своих потомков механизм использования диспинтерфейсов, необходимых для работы элементов управления ActiveX.

Кроме этого, он содержит все базовые интерфейсы ActiveX, перечисленные в табл. 26.1.

Так как для создания экземпляра компонента ActiveX применяется фабрика класса, в конструкторе класса нет необходимости. Но при создании компонента будет вызван метод

```
procedure Initialize; override;
```

который вы можете перекрыть и использовать для установки начальных значений и инициализации вспомогательных объектов в собственном элементе управления ActiveX.

Класс предоставляет готовый механизм нотификации о действиях с компонентом. При необходимости изменить значение какого-либо свойства в ответ на действия пользователя необходимо удостовериться в том, что это свойство можно менять. Для этого предусмотрен перегружаемый метод

```
function PropRequestEdit(const PropertyName: WideString): Boolean;
overload;
function PropRequestEdit(DispID: TDispID): Boolean; overload;
```

В качестве параметра необходимо задать имя свойства `PropertyName` или его диспетчерский идентификатор `DispID`. Если свойство редактируемо, функция возвращает `True`.

При изменении значения какого-либо свойства компонента вы можете вызвать перегружаемый метод

```
procedure PropChanged(const PropertyName: WideString); overload;procedure
PropChanged(DispID: TDispID); overload;
```

передавая ему в качестве параметра имя свойства `PropertyName` или его диспетчерский идентификатор `DispID`. Метод сгенерирует событие `OnChanged`.

Примечание

При работе с перечисленными методами старайтесь использовать диспетчерские идентификаторы свойств, т. к. имена свойств обрабатываются внутри методов класса `TActiveXControl` и все равно преобразуются в идентификаторы.

После создания экземпляра компонента вы можете получить доступ к экземпляру класса VCL, имплементирующего ваш компонент ActiveX. Для этого используется свойство

```
property Control: TWinControl;
```

А для получения указателей на необходимые интерфейсы можно вызвать метод

```
function ObjQueryInterface(const IID: TGUID; out Obj): HResult; override;
```

Обратите внимание, что это просто реализация соответствующего метода интерфейса `IUnknown`, который, конечно же, поддерживается классом `TActiveXControl`.

Фабрика класса компонента ActiveX

Как уже упоминалось, экземпляры компонентов ActiveX в Delphi создаются с помощью фабрики класса. Ее функциональность унаследована из аналогичной фабрики технологии Автоматизация. Фабрика класса для компонентов ActiveX инкапсулирована в классе `TActiveXControlFactory`.

Среда разработки Delphi как контейнер ActiveX

Среда разработки Delphi является контейнером ActiveX.

При создании нового компонента ActiveX в Delphi с помощью мастера **ActiveX Control Wizard** для него автоматически создается специальный класс-оболочка, назначение которого — представлять все методы интерфейсов компонента ActiveX для взаимодействия со средой разработки. Таким образом реализован механизм адекватного представления и поведения элементов управления ActiveX в среде разработки Delphi.

Основные функции класса-надстройки реализованы в классе `TOleControl`. Его свойства и методы обеспечивают необходимое управление компонентом ActiveX. Обычно разработчику нет необходимости напрямую обращаться к методам этого класса.

Регистрация компонентов ActiveX

Для регистрации созданных в Delphi элементов управления ActiveX можно использовать возможности операционной системы или среды разработки.

В операционной системе в вашем распоряжении есть системная утилита regsvr32.exe

В Delphi можно применять утилиту tregsvr, которая поставляется с Delphi в исходных текстах (папка ..\Demos\ActiveX\TRegSvr) или специальные пункты меню.

Для регистрации компонента в качестве элемента управления ActiveX выберите команду **Register ActiveX Server** из меню **Run**.

Для разрегистрации элемента управления служит команда **Unregister ActiveX Server** из меню **Run**.

Использование готовых компонентов ActiveX

С Delphi поставляется ряд компонентов ActiveX, созданных сторонними разработчиками (`ChartFX`, `VSSpell` и др.), однако вам наверняка придется устанавливать и другие элементы управления, расширяя возможности среды. Опишем этот путь и те подводные камни, которые могут на нем встретиться.

Инсталляция готовых элементов управления ActiveX

Первым делом следует убедиться, что информация о нужных вам элементах ActiveX есть в реестре. Для этого нужно выбрать пункт меню **Component | Import Component**. Порядок импорта элемента управления ActiveX аналогичен импорту библиотеки типов, описанному в *главе 25*. В появившемся мастере необходимо сначала выбрать тип импортируемого компонента. В нашем случае это переключатель **Import ActiveX Control**. При переходе на следующую страницу в списке представлена информация обо всех элементах ActiveX, зарегистрированных в системе. Здесь необходимо выбрать нужный элемент управления ActiveX.

Как правило, все элементы ActiveX, распространяемые в составе крупных программных продуктов, регистрируются автоматически при инсталляции. Если вы скачали файл из Internet или взяли у знакомого, то зарегистрировать его можно по крайней мере тремя путями:

- использованием мастера **Import Component;**

- вызовом системной утилиты regsvr32.exe (входит в состав операционной системы);
- вызовом утилиты tregsvr, которая поставляется с Delphi в исходных текстах (папка ..\Demos\ActiveX\TRegSvr).

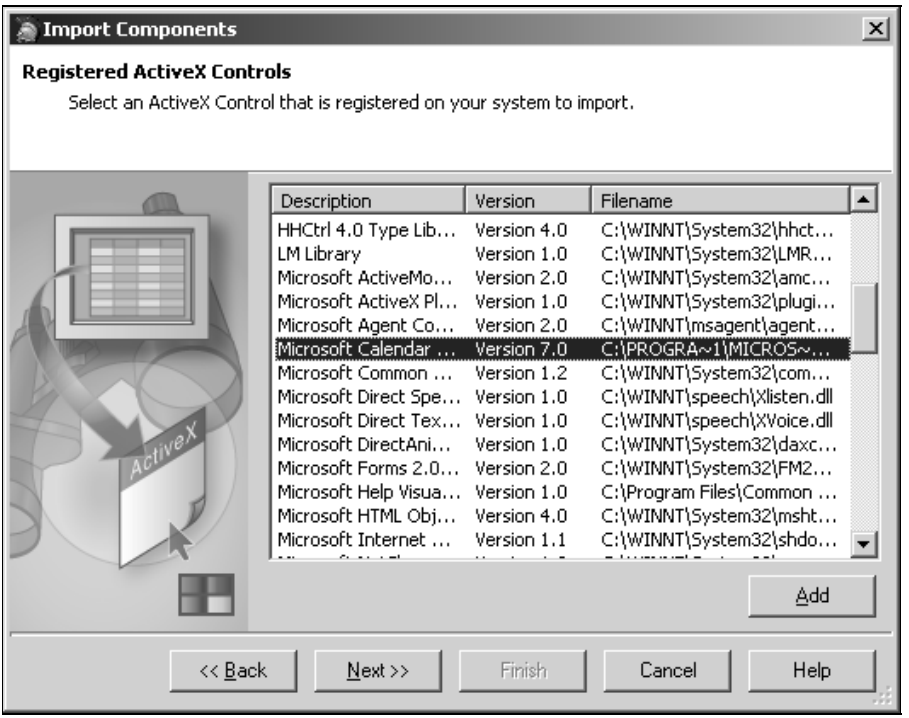


Рис. 26.3. Выбор элемента управления ActiveX в мастере импорта компонентов

После того как вы остановились на одном из элементов, в поле **Class Name(s)** будут перечислены все классы, содержащиеся в данном файле. В списке **Palette Page** вам предлагается выбрать "посадочную площадку" для будущего использования выбранного элемента в Delphi — страницу Палитры инструментов.

Если вы решили и дальше работать с этим элементом ActiveX, то следующий шаг — создание *файла представления* (wrapper). Этот файл представляет собой описание библиотеки типов — всех методов, свойств и событий, содержащихся в элементе, — на языке Delphi. Его имя складывается из имени ActiveX и строки `_TLB.PAS`. Нажатие кнопки **Create Unit** только создает этот файл (в целях ознакомления и тестирования), а нажатие кнопки **Install** продолжает процедуру установки.

Устанавливаемый элемент управления может быть помещен в модуль или проект. После компиляции пакета элемент ActiveX — в вашем распоряжении.

Примечание

Если вы будете проводить многочисленные эксперименты со всеми попавшимися под руку ActiveX, определяя степень их пригодности для вас, лучше создайте новый пакет. Также новый пакет понадобится, если вы уже определились с выбором и вам не нужна лишняя трата памяти.

Пример инсталляции элемента управления *TWebBrowser*

Рассмотрим в качестве примера элемент `TWebBrowser` из состава Microsoft Internet Controls (файл SHDOCVW.DLL). Он представляет собой ядро браузера от Microsoft и с большой долей вероятности уже имеется у вас на машине. К тому же его использование, во-первых, наглядно и, во-вторых, не лишено практического смысла.

После того как вы проделаете все описанные операции, в вашем распоряжении будет файл SHDOCVW_TLB.PAS. В нем среди прочего содержится описание интерфейса `IWebBrowser`. Помимо интерфейса, в этом же файле имеется описание объекта Delphi `TWebBrowser_V1`, который является надстройкой над интерфейсом `IWebBrowser` и содержит методы и свойства, в точности соответствующие методам и свойствам интерфейса.

Зачем это сделано?

Чтобы подвести единый базис под все встраиваемые компоненты и обеспечить им с точки зрения Delphi сходное поведение. В принципе, все классы-надстройки над ActiveX порождены от класса `TOleControl`. У него есть свойство `OleObject`, которое представляет собой ссылку на нужный вам интерфейс. Однако надежнее вызывать методы все же обычным путем — через методы соответствующего класса Delphi.

Методы интерфейса `IWebBrowser` обеспечивают выполнение основных операций для Internet-браузера. Не правда ли, все названия свойств и методов `IWebBrowser` просты и понятны? Интерфейс `IWebBrowser` описан в модуле SCHDocVv.pas, методы интерфейса представлены в листинге 26.1.

Листинг 26.1. Объявление интерфейса `IWebBrowser` в модуле SCHDocVv.pas

```
IWebBrowser = interface(IDispatch)
    ['{EAB22AC1-30C1-11CF-A7EB-0000C05BAE0B}']
    procedure GoBack; safecall;
    procedure GoForward; safecall;
```

```
    procedure GoHome; safecall;
    procedure GoSearch; safecall;
    procedure Navigate(const URL: WideString; var Flags: OleVariant;
var TargetFrameName: OleVariant; var PostData: OleVariant; var Headers:
OleVariant); safecall;
    procedure Refresh; safecall;
    procedure Refresh2(var Level: OleVariant); safecall;
    procedure Stop; safecall;
    function  Get_Application: IDispatch; safecall;
    function  Get_Parent: IDispatch; safecall;
    function  Get_Container: IDispatch; safecall;
    function  Get_Document: IDispatch; safecall;
    function  Get_TopLevelContainer: WordBool; safecall;
    function  Get_Type_: WideString; safecall;
    function  Get_Left: Integer; safecall;
    procedure Set_Left(pl: Integer); safecall;
    function  Get_Top: Integer; safecall;
    procedure Set_Top(pl: Integer); safecall;
    function  Get_Width: Integer; safecall;
    procedure Set_Width(pl: Integer); safecall;
    function  Get_Height: Integer; safecall;
    procedure Set_Height(pl: Integer); safecall;
    function  Get_LocationName: WideString; safecall;
    function  Get_LocationURL: WideString; safecall;
    function  Get_Busy: WordBool; safecall;
    property Application: IDispatch read Get_Application;
    property Parent: IDispatch read Get_Parent;
    property Container: IDispatch read Get_Container;
    property Document: IDispatch read Get_Document;
    property TopLevelContainer: WordBool read Get_TopLevelContainer;
    property Type_: WideString read Get_Type_;
    property Left: Integer read Get_Left write Set_Left;
    property Top: Integer read Get_Top write Set_Top;
    property Width: Integer read Get_Width write Set_Width;
    property Height: Integer read Get_Height write Set_Height;
    property LocationName: WideString read Get_LocationName;
    property LocationURL: WideString read Get_LocationURL;
    property Busy: WordBool read Get_Busy;
  end;
```

Теперь давайте применим полученные сведения о возможностях компонента ActiveX `TWebBrowser` на практике. Попробуем собрать собственный простой браузер за несколько минут.

Создадим новый проект — обычное приложение — и используем его форму. На форму перенесем компонент ActiveX `TWebBrowser` и применим простой рецепт.

Рецепт: форма, список выбора, несколько кнопок по вкусу (см. рис. 26.3). Кнопки могут соответствовать тем командам, которые вы привыкли видеть, к примеру, в Microsoft Internet Explorer.

Для кнопок из нашего примера создадим обработчики нажатия, поместив в них соответствующие методы из интерфейса `IWebBrowser`, предоставляемого компонентом ActiveX `TWebBrowser`.

Листинг 26.2 иллюстрирует код.

Листинг 26.2. Методы-обработчики событий для браузера на основе элемента управления `TWebBrowser`

```
procedure TForm1.BackToolButtonClick(Sender: TObject);
begin
 WebBrowser_V11.GoBack;
end;
procedure TForm1.GoToolButtonClick(Sender: TObject);
var ov1,ov2,ov3,ov4: OleVariant;
begin
 WebBrowser_V11.Navigate(ComboBox1.Text,ov1,ov2,ov3,ov4);
end;
procedure TForm1.FrwToolButtonClick(Sender: TObject);
begin
 WebBrowser_V11.GoForward;
end;
procedure TForm1.StopToolButtonClick(Sender: TObject);
begin
 WebBrowser_V11.Stop;
end;
procedure TForm1.HomeToolButtonClick(Sender: TObject);
begin
 WebBrowser_V11.GoHome;
end;
```

```
procedure TForm1.SrchToolButtonClick(Sender: TObject);
begin
 WebBrowser_V11.GoSearch;
end;
```

А вот и результат. После компиляции и запуска приложения мы получаем работоспособный браузер Internet с набором важнейших функций (рис. 26.4). При этом все, что мы сделали, — это связали методы компонента ActiveX с кнопками.

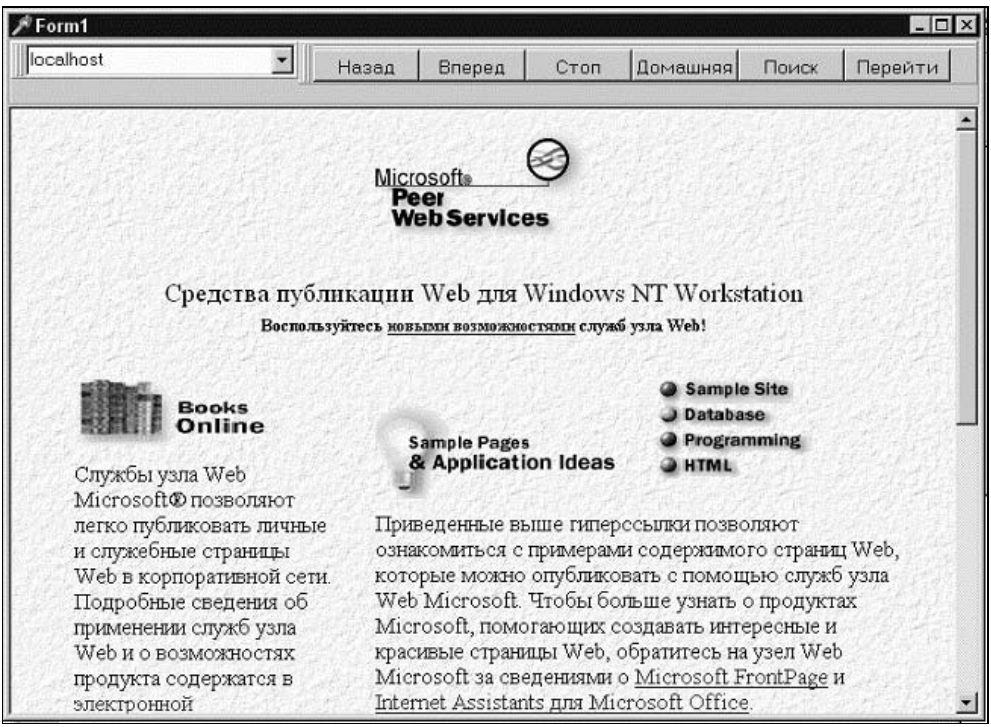


Рис. 26.4. Браузер, собранный на основе элемента управления ActiveX фирмы Microsoft

Приложив к этому примеру еще немного усилий, вы получите полноценный браузер. Он может пригодиться вам при отладке приложений, примеры которых рассматриваются далее в этой книге.

Резюме

Разработка элементов управления ActiveX базируется на возможностях технологий COM и Автоматизация.

Элементы управления ActiveX являются объектами COM и подчиняются общим для таких объектов правилам. Такие элементы могут функционировать лишь в составе внутреннего сервера (динамической библиотеки).

Для реализации элемента управления приложение должно обладать возможностями контейнера ActiveX. Существует ряд широко известных приложений, являющихся контейнерами. Среди них Microsoft Internet Explorer, Microsoft Visual Basic, Microsoft Word и Delphi.

ЧАСТЬ VII

РАСПРЕДЕЛЕННЫЕ ПРИЛОЖЕНИЯ БАЗ ДАННЫХ

Глава 27

Технология Data Snap

Ранее в этой книге мы рассматривали вопросы создания обычных приложений БД, работающих с базами данных на локальных компьютерах или в пределах локальной сети. Однако как быть, если необходимо создать приложение, которое может с одинаковым успехом работать как в локальной сети, так и с большим числом удаленных компьютеров.

Очевидно, что в этом случае модель доступа к данным должна быть расширена, т. к. наличие большого числа удаленных клиентов делает традиционные схемы создания приложений БД малоэффективными.

В этой главе мы рассмотрим модель распределенного приложения БД, которая называется многозвенной (multitiered), и, в частности, ее наиболее простой вариант — трехзвенное распределенное приложение. Частями такого приложения являются:

- собственно сервер базы данных;
- сервер приложения (ПО промежуточного слоя);
- клиентская часть приложения.

Все они объединены в единое целое механизмом взаимодействия (транспортный уровень) и обработки данных (уровень бизнес-логики).

Обобщая трехзвенную модель, следует отметить, что увеличение числа звеньев не затрагивает сервер БД и клиентское приложение. Дополнительные звенья усложняют только промежуточный слой, который может включать, например, сервер безопасности, сервер транзакций и т. д.

Компоненты и объекты Delphi, обеспечивающие разработку многозвенных приложений, объединены общим названием DataSnap.

Палитра инструментов Delphi содержит специальную категорию **DataSnap**, в которой доступно большинство рассматриваемых в главах этой части компонентов. Однако при разработке многозвенных приложений нам понадобятся и некоторые другие компоненты, которым также уделено достаточное внимание.

В этой главе рассматриваются следующие вопросы.

- Структура многозвенных приложений.
- Механизм удаленного доступа к данным DataSnap.
- Удаленные модули данных.
- Компоненты-провайдеры.
- Транспортные компоненты удаленных соединений DataSnap.
- Вспомогательные компоненты-брокеры соединений.

Структура многозвенного приложения в Delphi

Многозвенная архитектура приложений баз данных вызвана к жизни необходимостью обрабатывать на стороне сервера запросы от большого числа удаленных клиентов. Казалось бы, с этой задачей вполне могут справиться и приложения "клиент-сервер", основные элементы которых представлены в предыдущей части.

Однако при большом числе клиентов вся вычислительная нагрузка ложится на сервер БД, который обладает довольно скудным набором средств для реализации сложной бизнес-логики (хранимые процедуры, триггеры, просмотры и т. д.). И разработчики вынуждены существенно усложнять программный код клиентского ПО, а это крайне нежелательно при наличии большого числа удаленных клиентских компьютеров. Ведь с усложнением клиентского ПО возрастает вероятность ошибок и его обслуживание затруднено.

Многозвенная архитектура приложений БД призвана исправить перечисленные недостатки. Многозвенное приложение БД состоит из (рис. 27.1):

- "тонких" клиентских приложений, обеспечивающих лишь передачу, представление, редактирование и простейшую обработку данных;
- одного или нескольких звеньев ПО промежуточного слоя (сервер приложения), которые могут функционировать как на одном компьютере, так и распределенно (в локальной сети);
- сервера БД (Oracle, Sybase, MS SQL, InterBase и т. д.), поддерживающего функционирование базы данных и обрабатывающего запросы.

Итак, в рамках этой архитектуры, *"тонкие" клиенты* представляют собой простейшие приложения, обеспечивающие лишь передачу данных, их локальное кэширование, представление средствами пользовательского интерфейса, редактирование и простейшую обработку.

Клиентские приложения обращаются не к серверу БД напрямую, а к специализированному ПО промежуточного слоя. Это может быть и одно звено (простейшая трехзвенная модель), и более сложная структура.

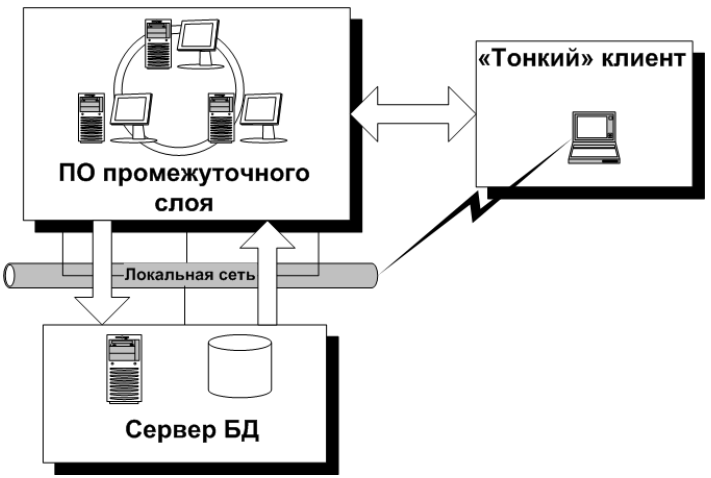


Рис. 27.1. Многозвенная архитектура приложений БД

ПО промежуточного слоя принимает запросы клиентов, обрабатывает их в соответствии с запрограммированными правилами бизнес-логики, при необходимости преобразует в форму, удобную для сервера БД, и отправляет серверу.

Сервер БД выполняет полученные запросы и отправляет результаты серверу приложений, который адресует данные клиентам.

Более простая трехзвенная модель содержит следующие элементы:

- "тонкие" клиенты;
- сервер приложения;
- сервер БД.

Далее мы будем рассматривать именно трехзвенную модель. В среде разработки Delphi имеется набор инструментов и компонентов для создания клиентского ПО и ПО промежуточного слоя. Серверная часть описывается в *главе 28*, вопросы создания клиентского ПО — в *главе 29*.

Сервер приложения взаимодействует с сервером БД посредством одной из технологий доступа к данным, реализованным в Delphi *(см. часть V)*. Это технологии dbGo, BDE, InterBase Express и dbExpress. Разработчик может выбрать наиболее подходящую, исходя из поставленной задачи и параметров сервера БД.

Для обеспечения работы с клиентами сервер приложения должен быть разработан на базе одной из следующих технологий, поддерживающих распределенный доступ:

- Автоматизация;
- Web;
- SOAP.

Удаленные клиентские приложения создаются с использованием специального набора компонентов, объединенных общим названием DataSnap. Эти компоненты инкапсулируют стандартные транспорты (DCOM, HTTP, сокеты) и обеспечивают соединение удаленного клиентского приложения с сервером приложения. Также компоненты DataSnap обеспечивают доступ клиента к функциям сервера приложений за счет использования интерфейса `IAppServer` *(см. главу 28)*.

Важную роль при разработке клиентских приложений играет компонент, инкапсулирующий клиентский набор данных. Его реализации также зависят от технологий доступа к данным и рассматриваются в *главе 29*.

Наряду с перечисленными преимуществами, наличие дополнительного звена — сервера приложений — дает некоторые дополнительные бонусы, которые могут быть весьма существенным подспорьем с точки зрения повышения надежности и эффективности системы.

Так как зачастую клиентские компьютеры — это достаточно слабые машины, реализация сложной бизнес-логики на стороне сервера позволяет существенно повысить быстродействие системы в целом. И не только за счет более мощной техники, но и за счет оптимизации выполнения однородных запросов пользователей.

Например, при чрезмерной загрузке сервера сервер приложения может самостоятельно обрабатывать запросы пользователей (ставить их в очередь или отменять) без дополнительной загрузки сервера БД.

Наличие сервера приложений повышает безопасность системы, т. к. вы можете организовать здесь авторизацию пользователей, да и любые другие функции безопасности, без прямого доступа к данным.

Кроме того, вам доступны защищенные каналы передачи данных, например, HTTPS.

Трехзвенное приложение в Delphi

Теперь рассмотрим составные части трехзвенного распределенного приложения в Delphi (рис. 27.2). Как уже говорилось, в Delphi целесообразно разрабатывать клиентскую часть трехзвенного приложения и ПО промежуточного слоя (сервер приложения).

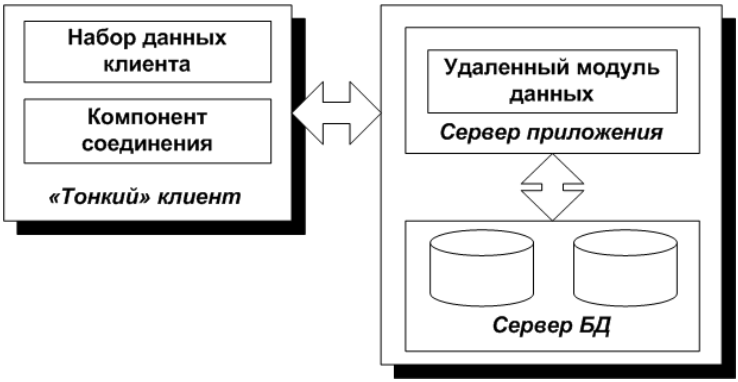


Рис. 27.2. Схема трехзвенного распределенного приложения

Части трехзвенных приложений разрабатываются с использованием компонентов DataSnap, а также некоторых других специализированных компонентов, в основном обеспечивающих функционирование клиента. Для доступа к данным применяется одна из технологий, реализованных в Delphi *(см. часть V)*.

Для передачи данных между сервером приложений и клиентами имеется интерфейс `IAppServer`, предоставляемый сервером приложения. Этот интерфейс используют компоненты-провайдеры `TDataSetProvider` на стороне сервера и компоненты `TClientDataSet` на стороне клиента.

Рассмотрим теперь части трехзвенного приложения более детально.

Сервер приложения

Сервер приложения инкапсулирует большую часть бизнес-логики распределенного приложения и обеспечивает доступ клиентов к базе данных.

С точки зрения разработчика основной частью сервера приложения является *удаленный модуль данных*. Давайте попробуем разобраться, почему.

Во-первых, удаленный модуль данных, в зависимости от реализации, инкапсулирует готовый удаленный сервер, который требует лишь настройки нескольких параметров и регистрации.

В состав Delphi входят несколько типов удаленных модулей данных.

Во-вторых, удаленный модуль данных обеспечивает взаимодействие с клиентами. Он предоставляет клиентскому приложению методы специального интерфейса `IAppServer` или его потомка. При помощи методов этого интерфейса организован процесс передачи и получения пакетов данных для клиентского приложения.

В-третьих, подобно обычному модулю данных *(см. главу 15)* удаленный модуль данных является платформой для размещения невизуальных компонентов доступа к данным и компонентов-провайдеров. Размещенные на нем компоненты соединений, транзакций и компоненты, инкапсулирующие наборы данных, обеспечивают трехзвенное приложение связью с сервером БД. Это могут быть наборы компонентов для различных технологий доступа к данным.

Помимо удаленного модуля данных, неотъемлемой частью сервера приложения являются компоненты-провайдеры `TDataSetProvider`. С каждым компонентом, инкапсулирующим набор данных, предназначенным для передачи клиенту, в модуле данных должен быть связан компонент-провайдер.

Для этого удаленный модуль данных должен содержать необходимое число компонентов-провайдеров `TDataSetProvider`. Эти компоненты передают пакеты данных клиентскому приложению, а точнее компонентам `TClientDataSet`, а также обеспечивают доступ к методам собственного интерфейса `IProviderSupport`. За счет методов этого интерфейса можно управлять пакетами пересылаемых данных на низком уровне.

Обычно такая задача перед разработчиками не встает. Просто нужно знать, что компоненты, работающие с данными, как на стороне клиента, так и сервера, используют этот интерфейс. Впрочем, если вы собрались создавать собственный вариант технологии DataSnap, описание интерфейса `IProviderSupprot` вам пригодится *(см. главу 28)*.

Клиентское приложение

Клиентское приложение в трехзвенной модели должно обладать лишь минимально необходимым набором функций, делегируя большинство операций по обработке данных серверу приложения.

В первую очередь, удаленное клиентское приложение должно обеспечить соединение с сервером приложения. Для этого предназначены компоненты соединений DataSnap:

- `TDCOMConnection` — использует DCOM;
- `TSocketConnection` — сокеты Windows;
- `TWebConnection` — HTTP.

Компоненты соединения DataSnap предоставляют интерфейс `IAppServer`, используемый компонентами-провайдерами на стороне сервера и компонентами `TClientDataSet` на стороне клиента для передачи пакетов данных.

Для работы с наборами данных служат компоненты `TClientDataSet`, функционирующие в режиме кэширования данных.

Для представления данных и создания пользовательского интерфейса в клиентском ПО применяются стандартные компоненты из категории **Data Controls** Палитры инструментов.

Подробнее о разработке клиентского ПО для распределенных многозвенных приложений БД рассказывается в *главе 28*.

Механизм удаленного доступа к данным DataSnap

Для передачи пакетов данных между компонентом-провайдером и клиентским набором данных (см. рис. 27.2) между клиентом и сервером должен существовать некий транспортный канал, обеспечивающий физическую передачу данных. Для этого существуют разнообразные транспортные протоколы, поддерживаемые операционной системой.

Различные типы соединений, позволяющие настроить транспорт и начать передачу и прием данных, инкапсулированы в нескольких компонентах DataSnap. Для создания соединения на основе того или иного транспортного протокола на стороне клиента разработчику достаточно перенести соответствующий компонент на форму и правильно настроить несколько свойств.

Этот компонент умеет взаимодействовать с однотипным удаленным модулем данных, который входит в состав сервера приложения.

Далее рассматриваются варианты настройки транспортных протоколов для компонентов соединения, использующих технологии DCOM, сокеты TCP/IP и HTTP.

Компонент *TDCOMConnection*

Компонент `TDCOMConnection` предоставляет транспорт на основе технологии Distributed COM и применяется в основном для организации соединения в рамках локальной сети.

Для настройки соединения DCOM в первую очередь необходимо задать имя компьютера, на котором функционирует сервер приложения. Для компонента `TDCOMConnection` это должен быть зарегистрированный сервер Автоматизации. Имя компьютера задается свойством

```
property ComputerName: string;
```

Если оно задано правильно, в списке свойства

```
property ServerName: string;
```

в Инспекторе объектов можно выбрать один из доступных серверов.

При выборе сервера также автоматически заполняется свойство

```
property ServerGUID: string;
```

содержащее глобальный идентификатор зарегистрированного сервера Автоматизации.

Причем для успешного соединения клиента с сервером приложений оба свойства должны быть заданы в обязательном порядке. Только имя сервера или только его GUID не обеспечат правильный доступ к удаленному объекту COM.

Открытие и закрытие соединения осуществляются свойством

```
property Connected: Boolean;
```

или методами

```
procedure Open;
procedure Close;
```

соответственно.

Для организации передачи данных между клиентом и сервером компонент `TDCOMConnection` предоставляет интерфейс `IAppServer`

```
property AppServer: Variant;
```

который также может быть получен методом

```
function GetServer: IAppServer; override;
```

Свойство

```
property ObjectBroker: TCustomObjectBroker;
```

позволяет использовать экземпляр компонента `TSimpleObjectBroker` для получения списка доступных серверов во время выполнения.

Методы-обработчики компонента `TDCOMConnection` представлены в табл. 27.1.

Таблица 27.1. *Методы-обработчики событий компонента* `TDCOMConnection`

Объявление	Описание
`property AfterConnect: TNotifyEvent;`	Вызывается после установления соединения
`property AfterDisconnect: TNotifyEvent;`	Вызывается после разрыва соединения
`property BeforeConnect: TNotifyEvent;`	Вызывается перед установлением соединения
`property BeforeDisconnect: TNotifyEvent;`	Вызывается перед разрывом соединения

Таблица 27.1 *(окончание)*

Объявление	Описание
`type TGetUsernameEvent = procedure(Sender: TObject; var Username: string) of object;` `property OnGetUsername: TGetUsernameEvent;`	Вызывается непосредственно перед появлением диалога удаленной авторизации пользователя. Для этого свойство `LoginPrompt` должно иметь значение `True`. Параметр `Username` может содержать имя пользователя по умолчанию, которое появится в диалоге
`type TLoginEvent = procedure(Sender:TObject; Username, Password: string) of object;` `property OnLogin: TLoginEvent;`	Вызывается после открытия соединения, если свойство `LoginPrompt` имеет значение `True`. Параметры `Username` и `Password` содержат имя пользователя и пароль, введенные при авторизации

Компонент *TSocketConnection*

Компонент `TSocketConnection` обеспечивает соединение клиента с сервером приложения за счет сокетов TCP/IP. Для успешного открытия соединения на стороне сервера должен работать сокет-сервер (приложение ScktSrvr.exe, рис. 27.3).

Для успешного соединения свойство

```
property Host: String;
```

должно содержать имя компьютера сервера. Дополнительно свойство

```
property Address: String;
```

должно содержать IP-адрес сервера.

Для открытия соединения должны быть заданы оба этих свойства.

Свойство

```
property Port: Integer;
```

устанавливает номер порта. По умолчанию это порт 211, но разработчик волен изменить порт, например, для различных категорий пользователей или для создания защищенного канала.

После правильного выбора компьютера в списке свойства

```
property ServerName: string;
```

в Инспекторе объектов появляется перечень доступных серверов Автоматизации, и после выбора сервера свойство

```
property ServerGUID: string;
```

которое содержит GUID зарегистрированного сервера, задается автоматически, хотя его можно задать и вручную.

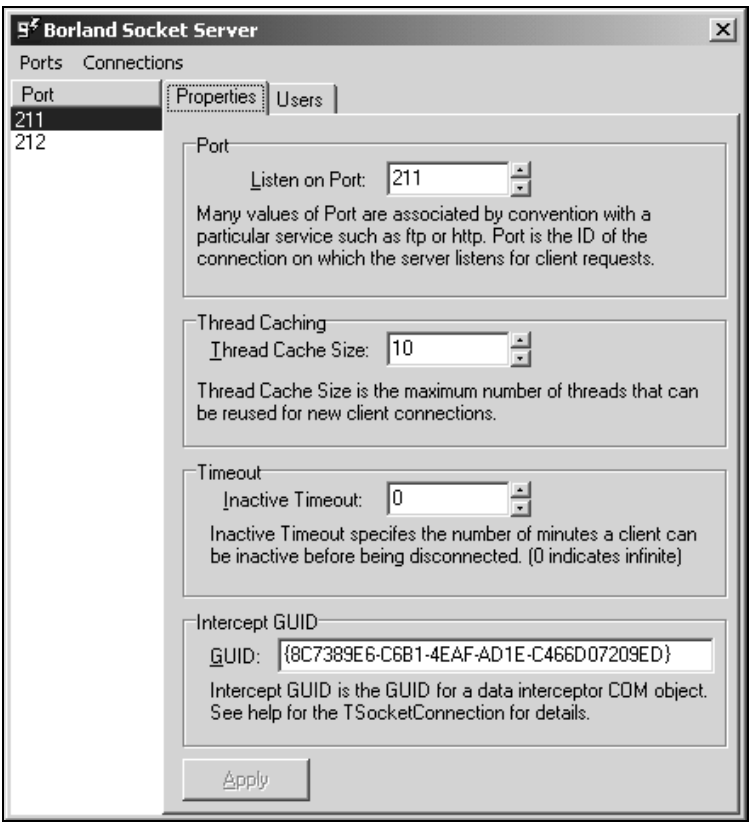


Рис. 27.3. Сокет-сервер ScktSrvr.exe

Метод

```
function GetServerList: OleVariant; virtual;
```

возвращает список зарегистрированных серверов Автоматизации.

Открытие и закрытие соединения осуществляется свойством

```
property Connected: Boolean;
```

или методами

```
procedure Open;
procedure Close;
```

соответственно.

Канал сокета TCP/IP может быть зашифрован. Для этого предназначено свойство

```
property InterceptName: string;
```

содержащее программный идентификатор объекта COM, обеспечивающего шифрование/дешифрирование данных в канале, и свойство

```
property InterceptGUID: string;
```

содержащее GUID этого объекта.

Этот объект COM перехватывает данные в канале и осуществляет их обработку, предусмотренную собственным программным кодом. Это могут быть шифрование, сжатие, обработка шумов и т. д.

Примечание

Создание объекта COM, обеспечивающего дополнительную обработку данных в канале, ложится на плечи разработчика. Объект-перехватчик должен поддерживать стандартный интерфейс `IDataIntercept`.

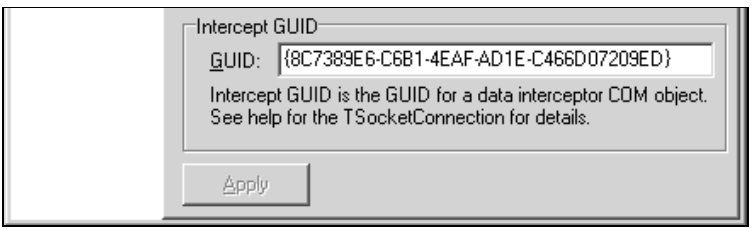


Рис. 27.4. Регистрация объекта-перехватчика COM в сокет-сервере

Естественно, на стороне сервера должен быть зарегистрирован объект COM, выполняющий обратную операцию. Для этого также применяется сокет-сервер (рис. 27.4). Строка **Interceptor GUID** на странице должна содержать GUID объекта-перехватчика COM.

Метод

```
function GetInterceptorList: OleVariant; virtual;
```

возвращает список зарегистрированных на сервере объектов-перехватчиков.

Для организации передачи данных между клиентом и сервером компонент `TSocketConnection` предоставляет интерфейс `IAppServer`

```
property AppServer: Variant;
```

который также может быть получен методом

```
function GetServer: IAppServer; override;
```

Свойство

```
property ObjectBroker: TCustomObjectBroker;
```

позволяет вызвать экземпляр компонента `TSimpleObjectBroker` для получения списка доступных серверов во время выполнения.

Методы-обработчики событий компонента `TSocketConnection` полностью совпадают с методами-обработчиками компонента `TDCOMConnection` (см. табл. 27.1).

Компонент *TWebConnection*

Компонент `TWebConnection` предоставляет клиенту соединение на основе транспорта HTTP. Для работы компонента на клиентском компьютере должна быть зарегистрирована библиотека wininet.dll. Обычно это не требует специальных усилий, т. к. данный файл уже имеется в системной папке Windows, если на компьютере установлен Internet Explorer.

На компьютере сервера должен быть инсталлирован Internet Information Server не ниже версии 4.0 или Netscape Enterprise не ниже версии 3.6. Перечисленное ПО обеспечивает доступ компонента `TWebConnection` к динамической библиотеке HTTPsrvr.dll, которая также должна находиться на сервере.

Например, если файл HTTPsrvr.dll расположен в папке Scripts IIS 4.0 на Web-сервере **www.someserver.com**, то свойство

```
property URL: string;
```

должно содержать следующее значение:

```
http://someserver.com/scripts/httpsrvr.dll
```

Если URL задан верно и сервер настроен правильно, то в списке свойства

```
property ServerName: string;
```

в Инспекторе объектов появляется перечень зарегистрированных серверов приложений. Имя одного из них должно содержаться в свойстве `ServerName`.

После выбора имени сервера в свойстве

```
property ServerGUID: string;
```

автоматически появляется GUID сервера.

Свойства

```
property UserName: string;
```

и

```
property Password: string;
```

при необходимости могут содержать имя и пароль пользователя, которые будут использованы при авторизации.

Свойство

```
property Proxy: string;
```

содержит имя используемого proxy-сервера.

В заголовок сообщений HTTP можно поместить имя приложения. Для этого предусмотрено свойство

```
property Agent: string;
```

Соединение открывается и закрывается при помощи свойства

```
property Connected: Boolean;
```

Аналогичные операции выполняют методы

```
procedure Open;
procedure Close;
```

Доступ к интерфейсу `IAppServer` предоставляет свойство

```
property AppServer: Variant;
```

или метод

```
function GetServer: IAppServer; override;
```

Список доступных соединению серверов приложений возвращает метод

```
function GetServerList: OleVariant; virtual;
```

Свойство

```
property ObjectBroker: TCustomObjectBroker;
```

позволяет использовать экземпляр компонента `TSimpleObjectBroker` для получения списка доступных серверов по время выполнения.

Методы-обработчики событий компонента `TWebConnection` полностью совпадают с методами-обработчиками компонента `TDCOMConnection` (см. табл. 27.1).

Вспомогательные компоненты-брокеры соединений

В состав компонентов DataSnap входит ряд дополнительных компонентов, облегчающих работу с соединениями удаленных клиентов с сервером приложений. Рассмотрим их.

Компонент *TSimpleObjectBroker*

Компонент `TSimpleObjectBroker` инкапсулирует список серверов, доступных для клиентов данного многозвенного распределенного приложения.

Список серверов создается на этапе разработки. При необходимости (отключение сервера, его перегрузка и т. д.) компонент соединения клиентского ПО может использовать один из запасных серверов из списка компонента `TSimpleObjectBroker` непосредственно во время выполнения.

Для этого необходимо заполнить список серверов компонента `TSimpleObjectBroker` и указать ссылку на него в свойстве `ObjectBroker` компонента соединения. И тогда при переоткрытии соединения имя сервера будет запрашиваться из списка компонента `TSimpleObjectBroker`.

Список серверов задается свойством

```
property Servers: TServerCollection;
```

На этапе разработки список серверов заполняется специализированным редактором (рис. 27.5), который вызывается при щелчке на кнопке свойства в Инспекторе объектов.

Рис. 27.5. Редактор списка серверов компонента `TSimpleObjectBroker`

Свойство `Servers` представляет собой коллекцию объектов класса `TServerItem`. Этот класс имеет несколько свойств, позволяющих описать основные параметры сервера.

Свойство

```
property ComputerName: string;
```

определяет имя компьютера, на котором функционирует сервер приложения. При этом можно также задать имя сервера для представления в списке серверов:

```
property DisplayName: String;
```

Запись о сервере можно сделать доступной или недоступной для выбора. Для этого имеется свойство

```
property Enabled: Boolean;
```

После неудачной попытки подключения с данной записью свойству

```
property HasFailed: Boolean;
```

присваивается значение `True`, и в дальнейшем эта запись не используется.

Свойство

```
property Port: Integer;
```

содержит номер порта для подключения к серверу.

В соединении значения этих свойств подставляются в соответствующие свойства компонента соединения:

```
DCOMConnection.ComputerName := TSimpleObjectBroker
(DCOMConnection.ObjectBroker).Servers[0].ComputerName;
```

Помимо списка серверов компонент `TSimpleObjectBroker` имеет лишь несколько вспомогательных свойств и методов.

Метод

```
function GetComputerForGUID(GUID: TGUID): string; override;
```

возвращает имя компьютера, на котором зарегистрирован сервер с GUID, заданным параметром `GUID`.

Метод

```
function GetComputerForProgID(const ProgID): string; override;
```

возвращает имя компьютера, на котором зарегистрирован сервер с именем, заданным параметром `ProgID`.

Свойство

```
property LoadBalanced: Boolean;
```

управляет выбором сервера из списка. При значении `True` запись о сервере выбирается случайным образом, иначе для соединения предлагается первая доступная запись о сервере.

Компонент *TLocalConnection*

Компонент `TLocalConnection` используется локально для получения доступа к существующим компонентам-провайдерам.

Свойство

```
property Providers[const ProviderName: string]: TCustomProvider;
```

содержит ссылки на все компоненты-провайдеры, размещенные с компонентом `TLocalConnection` в одном модуле данных. Индексация в списке осуществляется по имени компонента-провайдера.

Общее число компонентов-провайдеров в списке возвращает свойство

```
property ProviderCount: Integer;
```

Кроме этого, при помощи компонента `TLocalConnection` можно получить доступ к интерфейсу `IAppServer` локально через его свойство

```
property AppServer: IAppServer;
```

или метод

```
function GetServer: IAppServer; override;
```

Компонент *TSharedConnection*

Если интерфейс `IAppServer` удаленного модуля данных имеет метод, возвращающий ссылку на аналогичный интерфейс другого удаленного модуля данных, то первый модуль называется главным, а второй — дочерним *(см. главу 28)*. Компонент `TSharedConnection` служит для соединения клиентского приложения с дочерним удаленным модулем данных сервера приложения.

Свойство

```
property ParentConnection: TDispatchConnection;
```

должно содержать ссылку на компонент соединения с главным удаленным модулем данных сервера приложений. А дочерний удаленный модуль данных определяется свойством

```
property ChildName: string;
```

которое должно содержать его имя. Если интерфейс главного удаленного модуля данных настроен правильно, то в списке выбора свойства в Инспекторе объектов появляются имена всех дочерних удаленных модулей данных.

Интерфейс `IAppServer` дочернего удаленного модуля данных возвращает свойство

```
property AppServer: Variant;
```

или метод

```
function GetServer: IAppServer; override;
```

Методы-обработчики компонента `TSharedConnection` унаследованы от класса предка `TCustomConnection` (см. табл. 27.1).

Компонент *TConnectionBroker*

Компонент `TConnectionBroker` обеспечивает централизованное управление соединением клиентских наборов данных с сервером приложения. Для этого свойство `ConnectionBroker` клиентских наборов данных должно ссылаться на экземпляр компонента `TConnectionBroker`. Тогда для изменения соединения (например, при переходе с транспорта HTTP на сокеты TCP/IP) нет необходимости изменять значение свойства `RemoteServer` всех компонентов `TClientDataSet`, а достаточно изменить свойство

```
property Connection: TCustomRemoteServer;
```

компонента `TConnectionBroker`.

Доступ к интерфейсу `IAppServer` обеспечивает свойство

```
property AppServer: Variant;
```

или метод

```
function GetServer: IAppServer; override;
```

Методы-обработчики компонента `TConnectionBroker` полностью соответствуют табл. 27.1.

Резюме

Многозвенные распределенные приложения обеспечивают эффективное взаимодействие большого числа удаленных "тонких" клиентов с сервером БД при помощи ПО промежуточного слоя. Наиболее распространенной моделью является трехзвенная модель, где ПО промежуточного слоя состоит только из сервера приложения.

В Delphi для создания трехзвенных распределенных приложений предусмотрены компоненты DataSnap и удаленные модули данных. Все эти инструменты реализованы для различных типов транспортных протоколов.

Также в трехзвенных приложениях применяются компоненты-провайдеры `TDataSetProvider` и компоненты `TClientDataSet`, инкапсулирующие наборы данных на клиентской стороне.

Глава 28

Сервер приложения

Многозвенные распределенные приложения обеспечивают эффективный доступ удаленных клиентов к базе данных, т. к. в них для управления доступом к данным применяется специализированное ПО промежуточного слоя. В наиболее распространенной схеме — трехзвенном приложении — это сервер приложения, который выполняет следующие функции:

- обеспечивает авторизацию пользователей;
- принимает и передает запросы пользователей и пакеты данных;
- регулирует доступ клиентских запросов к серверу БД, балансируя нагрузку сервера БД;
- может содержать часть бизнес-логики распределенного приложения, обеспечивая существование "тонких" клиентов.

Delphi поддерживает разработку серверов приложений на основе ряда технологий:

- Web;
- Автоматизация;
- SOAP.

В этой главе рассматриваются следующие вопросы.

- Программные элементы сервера приложения Delphi.
- Структура сервера приложения.
- Типы удаленных модулей данных.
- Создание и настройка удаленных модулей данных.
- Роль компонентов-провайдеров в передаче данных клиентам.
- Методы интерфейса `IAppServer`.
- Регистрация сервера приложения.

Структура сервера приложения

Итак, сервер приложения является ПО промежуточного слоя трехзвенного распределенного приложения (рис. 28.1). Его основой является удаленный модуль данных. В Delphi предусмотрены удаленные модули данных нескольких типов.

Далее в этой главе мы детально рассмотрим вопросы использования удаленных модулей данных, инкапсулирующих функции серверов Автоматизации. Другие типы удаленных модулей данных рассматриваются в следующих частях этой книги.

Каждый удаленный модуль данных инкапсулирует интерфейс `IAppServer`, методы которого присутствуют в механизме удаленного доступа клиентов к серверу БД *(см. главу 27)*.

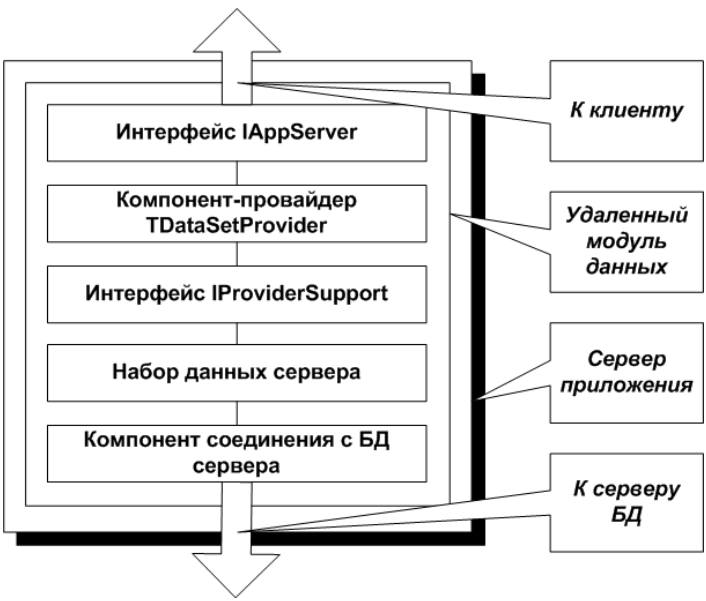


Рис. 28.1. Структура сервера приложения

Для обмена информацией с сервером БД модуль данных может содержать некоторое число компонентов доступа к данным (компонентов соединений и компонентов, инкапсулирующих набор данных).

Для обеспечения взаимодействия с клиентами удаленный модуль данных обязательно должен содержать необходимое количество компонентов `TDataSetProvider`, каждый из которых должен быть связан с соответствующим набором данных.

Внимание

Обмен данными сервера приложения с клиентами обеспечивает динамическая библиотека MIDAS.DLL, которая должна быть зарегистрирована на компьютере сервера приложения.

Для создания нового сервера приложения достаточно выполнить несколько простых операций.

1. Создать новый проект, выбрав в качестве типа проекта обычное приложение (пункт меню **File | New | VCL Forms Application - Delphi for Win32**) и сохранить его.
2. Выбрать из Репозитария Delphi значок удаленного модуля данных **Remote Data Module** из категории **Delphi Projects | ActiveX**.
3. Настроить параметры создаваемого удаленного модуля данных.
4. Разместить в удаленном модуле данных компоненты доступа к данным и настроить их. Здесь разработчик может выбрать один из имеющихся наборов компонентов *(см. часть V)* в зависимости от сервера БД и требуемых характеристик создаваемого приложения.
5. Разместить в удаленном модуле данных необходимое число компонентов `TDataSetProvider` и связать их с компонентами, инкапсулирующими наборы данных.
6. При необходимости создать для потомка интерфейса `IAppServer`, используемого в удаленном модуле данных дополнительные методы. Для этого создается новая библиотека типов.
7. Скомпилировать проект и создать исполняемый файл сервера приложения.
8. Зарегистрировать сервер приложения и при необходимости настроить дополнительное ПО.

Весь механизм удаленного доступа, инкапсулированный в удаленных модулях данных и компонентах-провайдерах, работает автоматически, без создания разработчиком дополнительного программного кода.

Далее в этой главе на простом примере рассматриваются все перечисленные этапы создания сервера приложения.

Интерфейс *IAppServer*

Интерфейс `IAppServer` является основной механизма удаленного доступа клиентских приложений к серверу приложения. Набор данных клиента использует его для общения с компонентом-провайдером на сервере приложения. Наборы данных клиента получают экземпляр `IAppServer` от компонента соединения в клиентском приложении (см. рис. 28.1).

При создании удаленных модулей данных каждому такому модулю ставится в соответствие вновь создаваемый интерфейс, предком которого является интерфейс `IAppServer`.

Разработчик может добавить к новому интерфейсу собственные методы, которые благодаря возможностям механизма удаленного доступа многозвенных приложений становятся доступны приложению-клиенту.

Свойство

```
property AppServer: Variant;
```

в клиентском приложении имеется как в компонентах удаленного соединения, так и клиентском наборе данных.

По умолчанию интерфейс является не сохраняющим состояние (stateless). Это означает, что вызовы методов интерфейса независимы и не привязаны к предыдущему вызову. Поэтому интерфейс `IAppServer` не имеет свойств, которые бы хранили информацию о состоянии между вызовами.

Обычно разработчику ни к чему вызывать методы интерфейса напрямую, однако его значение для многозвенных приложений трудно переоценить. И при детальной работе с механизмом удаленного доступа интерфейс понадобится так или иначе.

Методы интерфейса `IAppServer` представлены в табл. 28.1.

Таблица 28.1. *Методы интерфейса* `IAppServer`

Объявление	Описание
`function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer; var OwnerData: OleVariant): OleVariant; safecall;`	Передает изменения, полученные от клиентского набора данных, компоненту-провайдеру, определяемому параметром `ProviderName`. Изменения содержатся в параметре `Delta`. Параметр `MaxErrors` задает максимальное число ошибок, пропускаемых при сохранении данных перед прерыванием операции. Реальное число возникших ошибок возвращается параметром `ErrorCount`. Параметр `OwnerData` содержит дополнительную информацию, передаваемую между клиентом и сервером (например, значения параметров методов-обработчиков). Функция возвращает пакет данных, содержащий все записи, которые не были сохранены в базе данных по какой-либо причине

Таблица 28.1 *(окончание)*

Объявление	Описание
`function AS_DataRequest(const ProviderName: WideString; Data: OleVariant): OleVariant; safecall;`	Генерирует событие `OnDataRequest` для указанного провайдера `ProviderName`
`procedure AS_Execute(const ProviderName: WideString; const CommandText: WideString; var Params: OleVariant; var OwnerData: OleVariant); safecall;`	Выполняет запрос или хранимую процедуру, определяемые параметром `CommandText`, для провайдера, указанного параметром `ProviderName`. Параметры запроса или хранимой процедуры содержатся в параметре `Params`
`function AS_GetParams(const ProviderName: WideString; var OwnerData: OleVariant): OleVariant; safecall;`	Передает провайдеру `ProviderName` текущие значения параметров клиентского набора данных
`function AS_GetProviderNames: OleVariant; safecall;`	Возвращает список всех доступных провайдеров удаленного модуля данных
`function AS_GetRecords(const ProviderName: WideString, Count: Integer; out RecsOut: Integer; Options: Integer; const CommandText: WideString; var Params: OleVariant; var OwnerData:OleVariant): OleVariant; safecall;`	Возвращает пакет данных с записями набора данных сервера, связанного с компонентом-провайдером. Параметр `CommandText` содержит имя таблицы, текст запроса или имя хранимой процедуры, откуда необходимо получить записи. Но он работает только в случае, если для провайдера в параметре `Options` включена опция `poAllowCommandText`. Параметры запроса или процедуры помещаются в параметре `Params`. Параметр `count` задает требуемое число записей, начиная с текущей, если его значение больше нуля. Если параметр равен нулю, возвращаются только метаданные, если он равен –1, возвращаются все записи. Параметр `RecsOut` возвращает реальное число переданных записей
`function AS_RowRequest(const ProviderName: WideString; Row: OleVariant; RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;`	Возвращает запись набора данных, предоставляемого провайдером `ProviderName`, и определяемую параметром `Row`. Параметр `RequestType` содержит значение типа `TfetchOptions`

Большинство методов интерфейса имеют параметры `ProviderName` и `OwnerData`. Первый определяет имя компонента-провайдера, а второй содержит набор параметров, передаваемых в методы-обработчики.

Внимательный читатель обратил внимание, что метод `AS_GetRecords` подразумевает сохранение информации при работе интерфейса, поскольку он возвращает записи, начиная с текущей, хотя интерфейс `IAppServer` имеет тип `stateless`. Поэтому перед вызовом метода рекомендуется обновлять набор данных клиента.

Тип

```
TFetchOption = (foRecord, foBlobs, foDetails);
TFetchOptions = set of TFetchOption;
```

присутствует в параметре `RequestType` метода `AS_RowRequest`.

Назначение параметров здесь следующее:

- `foRecord` — возвращать значения полей текущей записи;
- `foBlobs` — возвращать значения полей типа BLOB текущей записи;
- `foDetails` — возвращать все подчиненные записи вложенных наборов данных для текущей записи.

Удаленные модули данных

Удаленный модуль данных является основой сервера приложения (см. рис. 28.1) для трехзвенного распределенного приложения.

Во-первых, удаленный модуль данных в зависимости от реализации инкапсулирует удаленный сервер.

Во-вторых, удаленный модуль данных инкапсулирует интерфейс `IAppServer`, обеспечивая тем самым выполнение функций сервера и обмен данными с удаленными клиентами.

В-третьих, он выполняет функции обычного модуля данных — на нем можно размещать компоненты доступа к данным.

Далее мы рассмотрим процесс создания сервера приложения на основе удаленного модуля данных `TRemoteDataModule`.

Удаленный модуль данных для сервера Автоматизации

Для создания удаленного модуля данных `TRemoteDataModule` удобен Репозитарий Delphi (команда меню **File | New | Other**): значок удаленного модуля данных **Remote Data Module** находится в категории **Delphi Projects | ActiveX**.

Перед созданием экземпляра удаленного модуля данных появляется диалог (рис. 28.2), в котором необходимо задать три параметра.

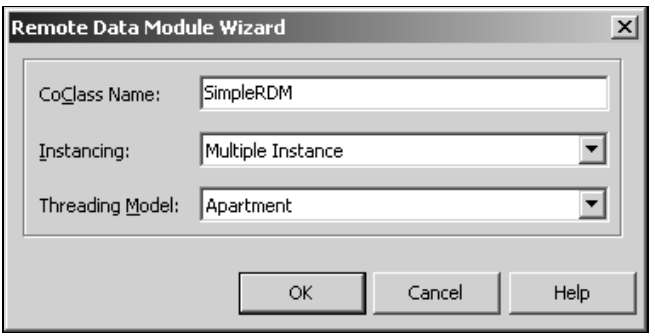


Рис. 28.2. Мастер создания удаленного модуля данных `TRemoteDataModule`

Строка **CoClass Name** должна содержать имя нового модуля данных, которое будет также совпадать с наименованием нового класса, создаваемого для поддержки нового модуля данных.

Список **Instancing** позволяет задать способ создания модуля данных:

- **Internal** — модуль данных обеспечивает функционирование лишь внутреннего сервера Автоматизации;
- **Single Instance** — для каждого клиентского соединения создается собственный экземпляр удаленного сервера Автоматизации в собственном процессе;
- **Multiple Instance** — для каждого клиентского соединения создается собственный экземпляр удаленного сервера Автоматизации в одном общем процессе.

Список **Threading Model** задает механизм обработки запросов клиентов:

- **Single** — поток запросов клиентов обрабатывается строго последовательно;
- **Apartment** — модуль данных одновременно обрабатывает один запрос. Однако если DLL для выполнения запросов создает экземпляры COM-объектов, то для запросов могут создаваться отдельные нити, в которых обработка ведется параллельно;
- **Free** — модуль данных может создавать нити для параллельного выполнения запросов;
- **Both** — аналогична модели Free, за исключением того, что все ответы клиентам возвращаются строго один за другим;

- **Neutral** — запросы клиентов могут направляться модулям данных в нескольких нитях одновременно. Используется только для технологии COM+.

При создании нового удаленного модуля данных создается специальный класс — наследник класса `TRemoteDataModule`. И фабрика класса на основе класса `TComponentFactory`

Примечание

Класс `TComponentFactory` представляет собой фабрику класса для компонентов Delphi, инкапсулирующих интерфейсы. Поддерживает интерфейс `IClassFactory`.

Создадим, например, удаленный модуль данных `SimpleRDM`. В мастере создания модуля данных в качестве способа создания выберем **Single Instance**, а **Free** — как модель обработки запросов. После выбора параметров и нажатия кнопки **OK** Delphi автоматически создаст для нового модуля данных исходный код, представленный в листинге 28.1.

Листинг 28.1. Исходный код нового удаленного модуля данных и его фабрики класса

```
type
  TSimpleRDM = class(TRemoteDataModule, ISimpleRDM)
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean; const ClassID,
ProgID: string); override;
  public
    { Public declarations }
  end;
implementation
{$R *.DFM}
class procedure TSimpleRDM.UpdateRegistry(Register: Boolean; const
ClassID, ProgID: string);
begin
  if Register then
  begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
```

```
  end else
  begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;
initialization
  TComponentFactory.Create(ComServer, TSimpleRDM,
    Class_SimpleRDM, ciMultiInstance, tmApartment);
end.
```

Обратите внимание, что заданные при создании параметры модуля данных использованы в фабрике класса `TComponentFactory` в секции `initialization`.

Примечание

Фабрика класса `TComponentFactory` обеспечивает создание экземпляров компонентов Delphi, поддерживающих интерфейсы.

Метод класса `UpdateRegistry` создается автоматически и обеспечивает регистрацию и аннулирование регистрации сервера Автоматизации. Если параметр `Register` имеет значение `True`, выполняется регистрация, иначе — отмена регистрации.

Разработчик не должен задействовать этот метод, т. к. его вызов осуществляется автоматически.

Одновременно с модулем данных создается и его интерфейс — потомок интерфейса `IAppServer`. Его исходный код содержится в библиотеке типов проекта сервера приложения. Для удаленного модуля данных `SimpleRDM` созданный интерфейс `ISimpleRDM` представлен в листинге 28.2. Для удобства из листинга удалены автоматически добавляемые комментарии.

Листинг 28.2. Вновь созданная библиотека типов для сервера приложения с исходным кодом интерфейса удаленного модуля данных

```
  LIBID_SimpleAppSrvr: TGUID = '{93577575-0F4F-43B5-9FBE-A5745128D9A4}';
  IID_ISimpleRDM: TGUID = '{E2CBEBCB-1950-4054-B823-62906306E840}';
  CLASS_SimpleRDM: TGUID = '{DB6A6463-5F61-485F-8F23-EC6622091908}';
type
  ISimpleRDM = interface;
  ISimpleRDMDisp = dispinterface;
```

```
  SimpleRDM = ISimpleRDM;
  ISimpleRDM = interface(IAppServer)
    ['{E2CBEBCB-1950-4054-B823-62906306E840}']
  end;
  ISimpleRDMDisp = dispinterface
    ['{E2CBEBCB-1950-4054-B823-62906306E840}']
    function  AS_ApplyUpdates(const ProviderName: WideString; Delta:
    OleVariant; MaxErrors: Integer; out ErrorCount: Integer; var
    OwnerData: OleVariant): OleVariant; dispid 20000000;
    function  AS_GetRecords(const ProviderName: WideString; Count:
    Integer; out RecsOut: Integer; Options: Integer; const CommandText:
    WideString; var Params: OleVariant; var OwnerData: OleVariant):
    OleVariant; dispid 20000001;
    function  AS_DataRequest(const ProviderName: WideString; Data:
    OleVariant): OleVariant; dispid 20000002;
    function  AS_GetProviderNames: OleVariant; dispid 20000003;
    function  AS_GetParams(const ProviderName: WideString; var OwnerData:
    OleVariant): OleVariant; dispid 20000004;
    function  AS_RowRequest(const ProviderName: WideString; Row:
    OleVariant; RequestType: Integer; var OwnerData: OleVariant):
    OleVariant; dispid 20000005;
    procedure AS_Execute(const ProviderName: WideString; const
    CommandText: WideString; var Params: OleVariant; var OwnerData:
    OleVariant); dispid 20000006;
  end;
  CoSimpleRDM = class
    class function Create: ISimpleRDM;
    class function CreateRemote(const MachineName: string): ISimpleRDM;
  end;
implementation
uses ComObj;
class function CoSimpleRDM.Create: ISimpleRDM;
begin
  Result := CreateComObject(CLASS_SimpleRDM) as ISimpleRDM;
end;
class function CoSimpleRDM.CreateRemote(const MachineName: string):
ISimpleRDM;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_SimpleRDM) as
ISimpleRDM;
end;
end.
```

Обратите внимание, что интерфейс `ISimpleRDM` является потомком интерфейса `IAppServer`, рассмотренного выше.

Так как удаленный модуль данных реализует сервер Автоматизации, дополнительно к основному дуальному интерфейсу `ISimpleRDM` автоматически создан интерфейс диспетчеризации `ISimpleRDMDisp`. При этом для интерфейса диспетчеризации созданы методы, соответствующие методам интерфейса `IAppServer`.

Класс `CoSimpleRDM` обеспечивает создание COM-объектов, поддерживающих использование интерфейса. Для него автоматически созданы два метода класса.

Метод

```
class function Create: ISimpleRDM;
```

необходим при работе с локальным и внутренним сервером (in process).

Метод

```
class function CreateRemote(const MachineName: string): ISimpleRDM;
```

используется в удаленном сервере.

Оба метода возвращают ссылку на интерфейс `ISimpleRDM`.

Теперь, если проект с созданным модулем данных сохранить и зарегистрировать, он станет доступен в удаленных клиентских приложениях как сервер приложения.

Однако пока он не предоставляет клиентскому приложению никаких наборов данных, т. к. удаленный модуль данных пока пуст.

После создания удаленный модуль данных становится платформой для размещения компонентов доступа к данным и компонентов-провайдеров, которые наряду с модулем данных реализуют основные функции сервера приложения.

Дочерние удаленные модули данных

Один сервер приложения может содержать несколько удаленных модулей данных, которые, например, выполняют различные функции или обращаются к разным серверам БД. В этом случае процесс разработки серверной части не претерпевает изменений. При выборе имени сервера в компоненте удаленного соединения на стороне клиента *(см. главу 29)* будут доступны имена всех удаленных модулей данных, включенных в состав сервера приложения.

Однако тогда для каждого модуля понадобится собственный компонент соединения. Если это нежелательно, можно выбрать компонент `TSharedConnection`,

(см. главу 27), но в этом случае в интерфейсы удаленных модулей данных необходимо внести изменения.

Для того чтобы несколько модулей данных были доступны в рамках одного удаленного соединения, необходимо выделить один главный модуль данных, а остальные сделать дочерними.

Рассмотрим, что же это означает для практики создания удаленных модулей данных. Суть идеи проста. Интерфейс главного модуля данных (разработчик назначает модуль главным, исходя из собственных соображений) должен содержать свойства, указывающие на интерфейсы всех других модулей данных, которые также необходимо использовать в рамках одного соединения на клиенте. Такие модули данных и называются дочерними.

Если такие свойства (свойство должно иметь атрибут "только для чтения") существуют, все дочерние модули данных будут доступны в свойстве `ChildName` компонента `TSharedConnection` *(см. главу 27)*.

Например, если дочерний удаленный модуль данных носит название `Secondary`, главный модуль данных должен содержать свойство `Secondary`:

```
ISimpleRDM = interface(IAppServer)
    ['{E2CBEBCB-1950-4054-B823-62906306E840}']
    function  Get_Secondary: Secondary; safecall;
    property Secondary: Secondary read Get_Secondary;
  end;
```

А реализация метода `Get_Secondary` выглядит так.

```
function TSimpleRDM.Get_Secondary: Secondary;
begin
 Result := FSecondaryFactory.CreateCOMObject(nil) as ISecondary;
end;
```

Как видите, в простейшем случае достаточно вернуть ссылку на вновь созданный дочерний интерфейс.

Полностью пример создания дочернего удаленного модуля данных рассматривается далее в этой главе.

Провайдеры данных

Компонент-провайдер `TDataSetProvider` представляет собой мост между набором данных сервера приложений и клиентским набором данных. Он обеспечивает формирование и передачу пакетов данных клиентскому приложению и прием от него сделанных изменений (рис. 28.3).

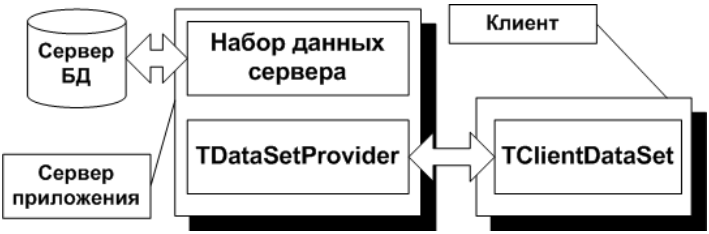


Рис. 28.3. Взаимодействие компонента-провайдера с клиентом

Все необходимые операции компонент выполняет автоматически. На стороне сервера разработчику необходимо лишь поместить компонент `TDataSetProvider` в удаленный модуль данных и затем связать его с набором данных сервера приложений. Для этого предназначено свойство

```
property DataSet: TDataSet;
```

Далее, если соединение в клиентском приложении настроено правильно, то в списке выбора свойства `ProviderName` компонента `TClientDataSet` в Инспекторе объектов появляются имена всех компонентов-провайдеров сервера приложения. Если связать клиентский набор данных с компонентом-провайдером, а затем открыть его, в клиентский набор данных будут переданы записи из набора данных сервера приложений, указанного в свойстве `DataSet` компонента-провайдера `TDataSetProvider`.

Компонент `TDataSetProvider` также содержит свойства, помогающие настроить процесс обмена данными.

Свойство

```
property ResolveToDataSet: Boolean;
```

управляет передачей данных от клиента серверу БД. Если оно имеет значение `True`, набор данных сервера приложений, заданный свойством `DataSet`, реагирует на все переданные изменения (изменяется состояние набора данных, вызываются методы-обработчики и т. д.). Иначе изменения направляются напрямую серверу БД. Если сервер приложений не должен отображать сделанные клиентом изменения, то свойству `ResolveToDataSet` можно присвоить значение `False`, что ускорит работу приложения.

Свойство

```
property Constraints: Boolean;
```

управляет передачей ограничений серверного набора данных клиентскому. Если свойство имеет значение `True`, ограничения передаются.

Свойство

```
property Exported: Boolean;
```

реализует в клиентском наборе данных интерфейс IAppServer. Для этого свойство должно иметь значение True.

Параметры компонента-провайдера задаются свойством

```
type
  TProviderOption = (poFetchBlobsOnDemand, poFetchDetailsOnDemand,
poIncFieldProps, poCascadeDeletes, poCascadeUpdates, poReadOnly,
poAllowMultiRecordUpdates, poDisableInserts, poDisableEdits,
poDisableDeletes, poNoReset, poAutoRefresh, poPropogateChanges,
poAllowCommandText, poRetainServerOrder);
  TProviderOptions = set of TProviderOption;
```

Набор параметров свойства задается присвоением элементам значения True

```
property Options: TProviderOptions;
```

Назначение параметров здесь следующее.

- poFetchBlobsOnDemand — включает передачу в клиентский набор данных значений полей типа BLOB. По умолчанию эта возможность отключена для ускорения работы.
- poFetchDetailsOnDemand — включает передачу в клиентский набор данных подчиненных записей для отношения "один ко многим". По умолчанию эта возможность отключена для ускорения работы.
- poIncFieldProps — включает передачу в клиентский набор данных нескольких свойств для объектов полей: Alignment, DisplayLabel, DisplayWidth, Visible, DisplayFormat, EditFormat, MaxValue, MinValue, Currency, EditMask, DisplayValues.
- poCascadeDeletes — включает автоматическое удаление подчиненных записей в отношении "один ко многим" на стороне сервера, если главная запись была удалена в клиентском наборе данных.
- poCascadeUpdates — включает автоматическое обновление подчиненных записей в отношении "один ко многим" на стороне сервера, если главная запись была изменена в клиентском наборе данных.
- poReadOnly — включает режим "только для чтения" для набора данных сервера.
- poAllowMultiRecordUpdates — включает режим внесения изменений сразу в несколько записей одновременно. Иначе все записи изменяются последовательно, одна за одной.
- poDisableInserts — запрещает клиенту вносить в набор данных сервера новые записи.
- poDisableEdits — запрещает клиенту вносить в набор данных сервера изменения.

- `poDisableDeletes` — запрещает клиенту удалять записи в наборе данных сервера.
- `poNoReset` — запрещает обновление набора данных сервера перед передачей записей клиенту (перед вызовом метода `AS_GetRecords` интерфейса `IAppServer`).
- `poAutoRefresh` — включает автоматическое обновление записей клиентского набора данных. По умолчанию эта возможность отключена для ускорения работы.
- `poPropogateChanges` — если в методах-обработчиках `BeforeUpdateRecord` или `AfterUpdateRecord` клиентского набора данных были сделаны дополнительные изменения, то после их записи в наборе данных сервера изменения снова направляются клиенту для обновления записи. Во включенном состоянии эта возможность позволяет полностью контролировать сохранение изменений на сервере.
- `poAllowCommandText` — позволяет изменять текст запроса SQL, имена хранимых процедур или таблиц в компоненте набора данных на сервере приложений.
- `poRetainServerOrder` — включает запрет на изменение порядка сортировки записей клиентом. Если этот параметр отключить, возможны ошибки отображения набора данных, проявляющиеся в появлении двойных записей.

Методы-обработчики компонента-провайдера данных представлены в табл. 28.2.

Таблица 28.2. *Методы-обработчики событий компонента* `TDataSetProvider`

Объявление	Описание
`Property AfterApplyUpdates: TremoteEvent;`	Вызывается после сохранения изменений, переданных от клиента, в наборе данных сервера
`Property AfterExecute: TRemoteEvent;`	Вызывается после выполнения запроса SQL или хранимой процедуры на сервере
`Property AfterGetParams: TremoteEvent;`	Вызывается после того, как компонент-провайдер сформировал набор параметров набора данных сервера для их передачи клиенту
`property AfterGetRecords: TRemoteEvent;`	Вызывается после того, как компонент-провайдер сформировал пакет данных для передачи набора данных сервера клиенту

Таблица 28.2 *(окончание)*

Объявление	Описание
`property AfterRowRequest: TRemoteEvent;`	Вызывается после обновления текущей записи клиента компонентом-провайдером
`property AfterUpdateRecord: TAfterUpdateRecordEvent;`	Вызывается сразу после обновления единичной записи на сервере
`property BeforeApplyUpdates: TRemoteEvent;`	Вызывается перед сохранением изменений, переданных от клиента, в наборе данных сервера
`property BeforeExecute: TRemoteEvent;`	Вызывается перед выполнением запроса SQL или хранимой процедуры на сервере
`property BeforeGetParams: TRemoteEvent;`	Вызывается перед тем, как компонент-провайдер сформировал набор параметров набора данных сервера для их передачи клиенту
`property BeforeGetRecords: TRemoteEvent;`	Вызывается перед тем, как компонент-провайдер сформировал пакет данных для передачи набора данных сервера клиенту
`property BeforeRowRequest: TRemoteEvent;`	Вызывается перед обновлением текущей записи клиента компонентом-провайдером
`property BeforeUpdateRecord: TBeforeUpdateRecordEvent;`	Вызывается непосредственно перед обновлением единичной записи на сервере
`property OnDataRequest: TDataRequestEvent;`	Вызывается при обработке запроса на получение данных клиентом
`property OnGetData: TProviderDataEvent;`	Вызывается после получения данных от набора данных сервера, но перед их отправкой клиенту
`property OnGetDataSetProperties: TGetDSProps;`	Вызывается при создании структуры параметров набора данных сервера для их передачи клиенту
`property OnGetTableName: TGetTableNameEvent;`	Вызывается при получении компонентом-провайдером имени таблицы, подлежащей обновлению
`property OnUpdateData: TProviderDataEvent;`	Вызывается при сохранении изменений в наборе данных сервера
`property OnUpdateError: TResolverErrorEvent;`	Вызывается при возникновении ошибки сохранения изменений в наборе данных сервера

Интерфейс *IProviderSupport*

Для обмена данными с набором данных на сервере компонент-провайдер применяет интерфейс `IProviderSupport`, который включен в любой компонент набора данных, произошедший от класса `TDataSet`. В зависимости от выбранной технологии доступа к данным каждый компонент, инкапсулирующий набор данных, имеет собственную реализацию методов интерфейса `IProviderSupport`.

Методы интерфейса могут понадобиться разработчику только при создании собственных компонентов, инкапсулирующих набор данных и наследующих от класса `TDataSet`.

Регистрация сервера приложения

Для того чтобы клиент мог "увидеть" сервер приложения, он должен быть зарегистрирован на компьютере сервера. В зависимости от технологии процесс регистрации имеет особенности.

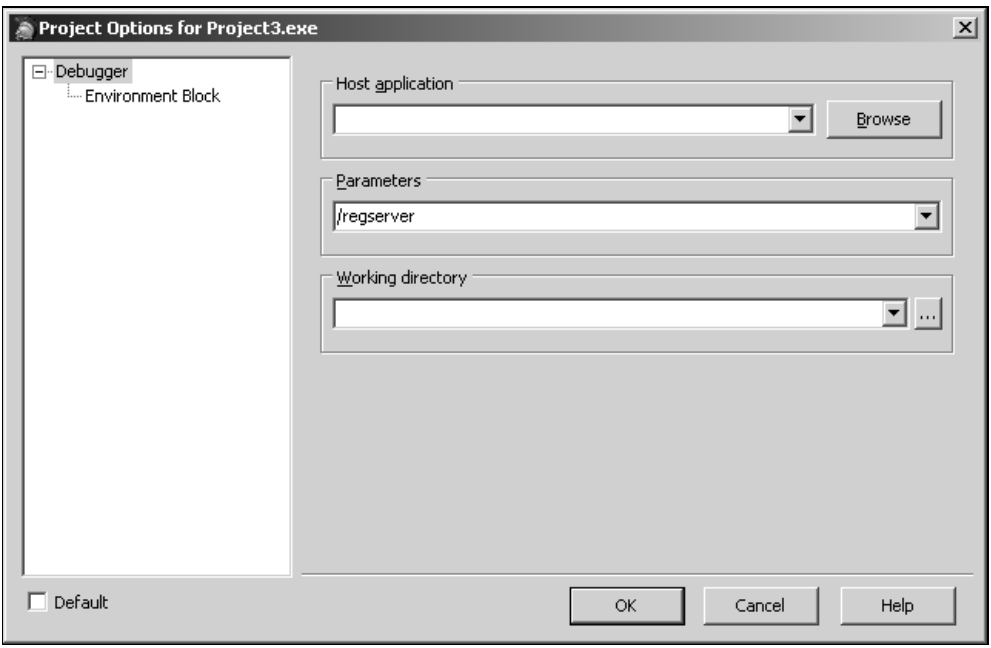


Рис. 28.4. Диалог параметров запуска приложения

Здесь же мы остановимся на регистрации сервера приложения с удаленным модулем данных `TRemoteDataModule` (сервер Автоматизации), который чрезвычайно прост.

Для исполняемых файлов достаточно запустить сервер с ключом `/regserver` или даже просто запустить исполняемый файл.

В среде разработки ключ можно поместить в диалоге команды меню **Run | Parameters**.

Для удаления регистрации применяется ключ `/unregserver`, но только в командной строке.

Для регистрации динамических библиотек — ключ `/regsvr32`.

Пример простого сервера приложения

В качестве примера рассмотрим процесс создания простого сервера приложения на основе удаленного модуля данных `TRemoteDataModule`. Для начала создадим новый проект — простое исполняемое приложение — и сохраним его под именем `SimpleAppSrvr`. Этот проект входит в состав группы проектов `SimpleRemote`, в нее впоследствии будет добавлено клиентское приложение. В табл. 28.3 приведен перечень файлов проекта `SimpleAppSrvr`.

Таблица 28.3. *Файлы проекта* `SimpleAppSrvr`

Файл	Назначение
uSimpleAppSrvr.pas	Стандартный файл проекта
SimpleAppSrvr_TLB.pas	Библиотека типов. Содержит объявления всех используемых в проекте интерфейсов
uSimpleRDM.pas	Файл главного удаленного модуля данных `SimpleRDM`
uSecondary.pas	Файл дочернего удаленного модуля данных `Secondary`

Примечание

Пример создания клиента для сервера приложения `SimpleAppSrvr` рассматривается в *главе 29*.

Главная форма сервера приложения

При первом запуске сервера приложения необходимо настроить соединение ADO, которое используется для доступа к источнику данных в модулях данных. В главной форме `fmMain` сервера приложения (файл uSimpleAppSrvr.pas) необходимо выбрать реальный путь к базе данных, нажав кнопку стандарт-

ного диалога выбора файла (рис. 28.5). Выбранный файл отображается в однострочном редакторе `edDataPath`.

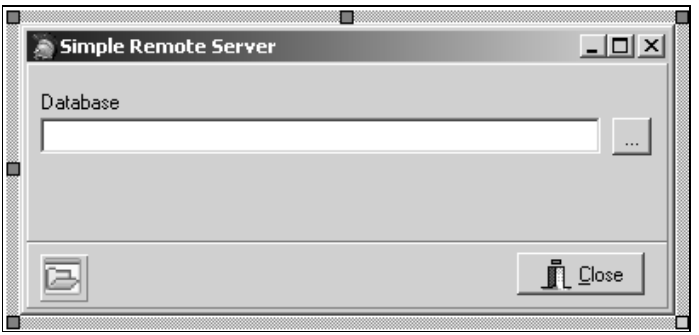


Рис. 28.5. Главная форма сервера приложения SimpleAppSrvr

В дальнейшем, в методах-обработчиках `BeforeConnect` компонентов `TADOConnection` в удаленных модулях данных путь к базе данных используется для настройки соединения.

Путь к базе данных сохраняется в файле SimpleApp.ini.

Главный удаленный модуль данных

Добавим в проект новый удаленный модуль данных, используя для этого Репозитарий Delphi. Затем в появившемся диалоге (см. рис. 28.2) зададим имя модуля — `SimpleRDM` и его параметры:

- способ создания — **Single Instance** — для каждого клиента создается собственный модуль данных;
- способ обработки запросов — **Free**.

Метод класса `UpdateRegistry` для модуля данных создается автоматически и обеспечивает регистрацию и аннулирование регистрации сервера Автоматизации (см. листинг 28.1).

Одновременно с удаленным модулем данных автоматически создается библиотека типов и в ней дуальный интерфейс `ISimpleRDM` и интерфейс диспетчеризации `ISimpleRDMDisp` (см. листинг 28.2).

Примечание

Для каждого вновь созданного интерфейса автоматически назначается GUID.

Разместим в модуле `SimpleRDM` компоненты для доступа к файлам демонстрационной базы данных Microsoft Access ..\Program Files\Common Files\

Borland Shared\Data\dbdemos.mdb. Доступ к данным реализуем через провайвер OLE DB. Это компонент `TADOConnection`, обеспечивающий соединение и три табличных компонента `TADOTable`, инкапсулирующие наборы данных из таблиц `Orders`, `Customer` и `Employee`.

Перед открытием соединения вызывается метод-обработчик

```
procedure TSimpleRDM.conMastAppBeforeConnect(Sender: TObject);
begin
 if fmMain.edDataPath.Text <> ''
  then conMastApp.ConnectionString := 'Provider = Micro-
soft.Jet.OLEDB.4.0;Persist Security Info=False;Data Source=' +
fmMain.edDataPath.Text
  else Abort;
end;
```

в котором настраивается свойство `ConnectionString`.

Свойство `LoginPrompt = False` запрещает отображение диалога регистрации при открытии соединения.

Каждый табличный компонент связан с компонентом-провайдером `TDataSetProvider`. Свойство провайдера `ResolveToDataSet = False` запрещает отображение изменений, полученных от клиента, в наборе данных связанного компонента. Вместо этого данные напрямую сохраняются в базе данных. Это увеличивает быстродействие приложения.

Дочерний удаленный модуль данных

Дополнительно к основному модулю данных создадим дочерний модуль данных Secondary. Для того чтобы связать главный модуль данных с дочерним, необходимо добавить к интерфейсу `ISimpleRDM` метод, возвращающий ссылку на интерфейс дочернего модуля данных. В нашем примере это метод `Get_Secondary`.

Для его создания воспользуемся библиотекой типов сервера (рис. 28.6).

В дереве в левой части окна выберем интерфейс `ISimpleRDM` и создадим для него новое свойство только для чтения, переименуем его в `Secondary`. Одновременно со свойством будет создан метод, обеспечивающий чтение свойства. Переименуем его в `Get_Secondary`. Метод должен возвращать тип `Secondary`. Для его установки воспользуемся списком **Type** на странице **Attributes** в правой панели окна библиотеки типов (см. рис. 28.6).

После обновления исходного кода библиотеки типов (кнопка **Refresh Implementation**) описание нового свойства и метода интерфейса `ISimpleRDM` появится в файле файл SimpleAppSrvr_TLB.PAS.

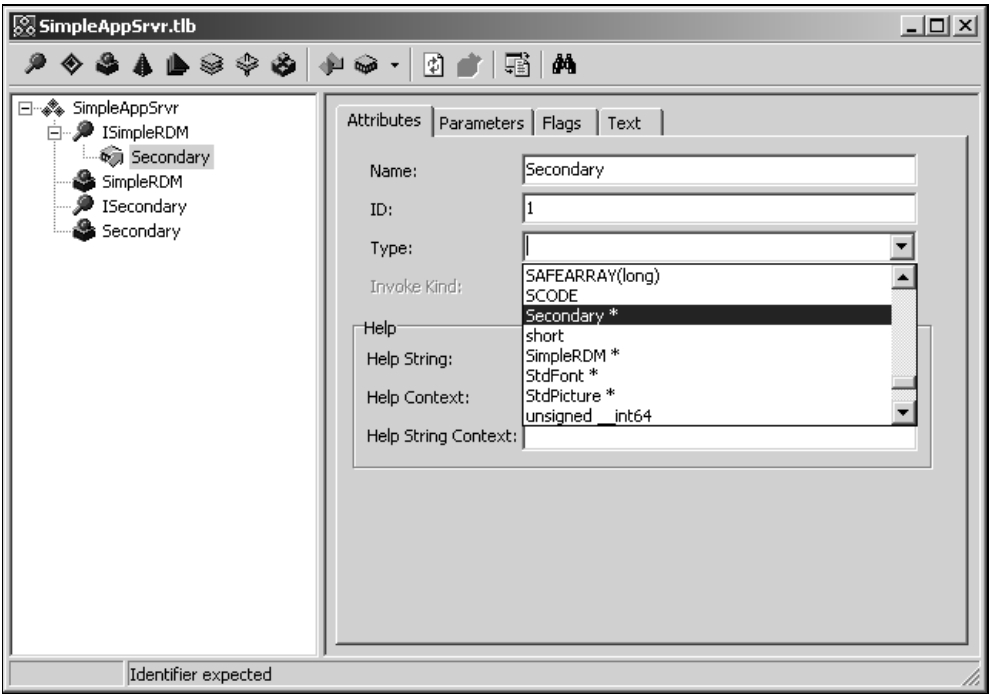


Рис. 28.6. Библиотека типов сервера приложения `SimpleAppSrvr`

Теперь объявление интерфейса `ISimpleRDM` выглядит так:

```
ISimpleRDM = interface(IAppServer)
    ['{E2CBEBCB-1950-4054-B823-62906306E840}']
    function Get_Secondary: Secondary; safecall;
    property Secondary: Secondary read Get_Secondary;
  end;
```

Одновременно в объявлении удаленного модуля данных `SimpleRDM` в файле uSimpleRDM.pas появится метод `Get_Secondary`. Его исходный код должен выглядеть следующим образом:

```
function TSimpleRDM.Get_Secondary: Secondary;
begin
 Result := FSecondaryFactory.CreateCOMObject(nil) as ISecondary;
end;
```

Теперь модуль данных `Secondary` стал дочерним для модуля `SimpleRDM`.

Модуль `Secondary` также содержит компоненты для доступа к базе данных Microsoft Access. База данных dbdemos.mdb поставляется вместе с Delphi. Соединение обеспечивается компонентом `TADOConnection`, который настраивается методом-обработчиком `TSecondary.conMastAppBeforeConnect`.

Два табличных компонента `TADOTable` инкапсулируют таблицы `Vendors` и `Parts` из базы данных dbdemos.mdb. Дополнительно между этими двумя компонентами установлено отношение "один ко многим". Свойство `MasterSource` компонента `tblParts` указывает на компонент `dsVendors` (класс `TDataSource`), связанный с компонентом `tblVendors`. Свойства `MasterFields` и `IndexFieldNames` компонента `tblParts` содержат имя общего для двух таблиц поля `VendorNo`.

Отношение "один ко многим", созданное для двух таблиц, позволит продемонстрировать в примере клиентского приложения работу с вложенными наборами данных *(см. главу 29)*.

Регистрация сервера приложения

Теперь, когда сервер приложения готов, остался последний этап — регистрация сервера. Для этого достаточно запустить исполняемый файл проекта на компьютере, который будет обеспечивать работу сервера приложения.

После регистрации сервер приложения доступен из всех клиентских приложений, компоненты соединения DataSnap настроены на компьютер сервера приложения.

Резюме

Сервер приложения представляет собой ПО промежуточного слоя для трехзвенных распределенных приложений. Он обеспечивает связь удаленных клиентов с сервером БД и реализует бóльшую часть бизнес-логики распределенного приложения.

В Delphi сервер приложения создается на основе удаленных модулей данных, реализация которых различается для различных технологий удаленного доступа. Удаленные модули данных имплементируют интерфейс `IAppServer`. Непосредственный доступ к данным обеспечивают компоненты-провайдеры `TDataSetProvider` при помощи интерфейса `IProviderSupport`.

Глава 29

Клиент многозвенного распределенного приложения

Клиентское ПО в распределенном многозвенном приложении имеет особенности архитектуры, продиктованные его ролью, ведь бóльшая часть бизнес-логики и функций обработки данных сосредоточена в сервере приложений *(см. главу 28)*. Такая схема призвана обеспечить более высокую эффективность обработки запросов многочисленных удаленных клиентов, а также упрощает обслуживание клиентского ПО. Клиенты, выполняющие лишь необходимый минимум операций, называются "тонкими".

Клиенты многозвенных приложений обеспечивают выполнение следующих функций:

- соединение с сервером приложений, прием и передача данных;
- отображение средствами пользовательского интерфейса;
- простейшие операции редактирования;
- сохранение локальных копий данных.

При разработке клиентских частей многозвенных приложений в Delphi используются компоненты DataSnap *(см. главу 27)*, а также компонент `TClientDataSet`, роль которого трудно переоценить.

Помимо новых компонентов, в процессе разработки применяются стандартные компоненты отображения данных, а также обычная схема связывания визуальных компонентов с набором данных через компонент `TDataSource` *(см. главу 15)*.

В этой главе рассматриваются следующие вопросы.

- Структура клиентского приложения.
- Соединение удаленного клиента с сервером приложений.
- Набор данных клиента в компоненте `TClientDataSet`, локальное кэширование данных.
- Основные операции обработки данных, выполняемые клиентским набором данных.

- Вложенные наборы данных.
- Обработка локальных ошибок клиентского набора данных и ошибок сервера приложений.

Структура клиентского приложения

По своей структуре клиентское приложение подобно обычному приложению баз данных, рассматриваемому в *главе 15*.

Соединение клиента с сервером приложений осуществляется специализированными компонентами DataSnap. В зависимости от потребностей в транспортной технологии разработчику доступны следующие компоненты:

- `TDCOMConnection` — DCOM;
- `TSocketConnection` — сокеты TCP/IP;
- `TWEBConnection` — Web.

Эти компоненты взаимодействуют с удаленным модулем данных, входящим в состав сервера, при помощи методов интерфейса `IAppServer`.

Также в клиентском приложении могут присутствовать дополнительные, определенные разработчиком, методы интерфейса удаленного модуля данных, унаследованного от интерфейса `IAppServer`. Подробнее об этих компонентах и способах их настройки на удаленный сервер приложения *см. главу 28*.

Внимание

Соединение с сервером приложений обеспечивает динамическая библиотека MIDAS.DLL, которая должна быть зарегистрирована на компьютере клиента.

Как и обычное приложение БД, клиент многозвенного распределенного приложения должен содержать компоненты, инкапсулирующие набор данных, которые связаны с визуальными компонентами отображения данных посредством компонентов `TDataSource` (рис. 29.1).

Очевидно, что набор данных сервера должен быть скопирован клиентским приложением в некий локальный буфер. При этом должен существовать эффективный механизм загрузки данных сравнительно небольшими порциями, что позволяет значительно разгрузить транспортный канал между клиентом и сервером приложений.

Кэширование и редактирование данных в клиентском приложении обеспечивает специализированный компонент `TClientDataSet`, отдаленным предком которого является класс `TDataSet`. Помимо унаследованных от предков методов, класс `TClientDataSet` инкапсулирует ряд дополнительных функций, облегчающих управление данными.

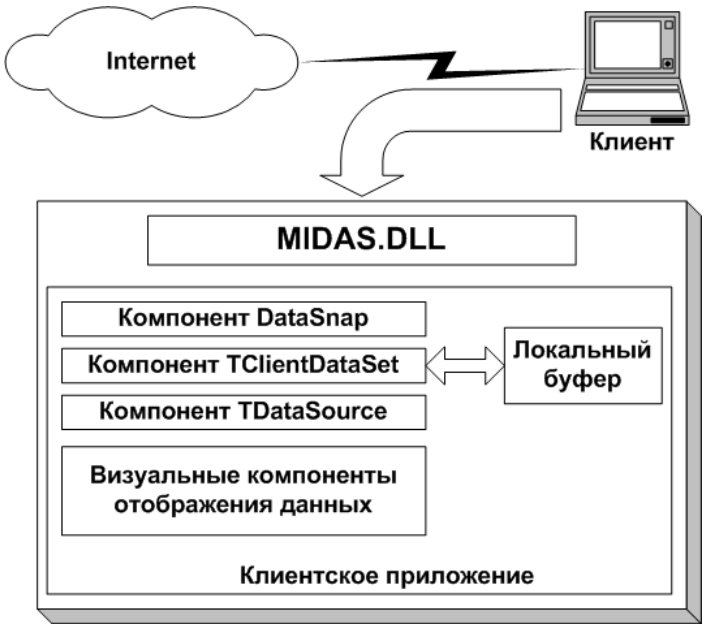


Рис. 29.1. Структура клиентской части многозвенного приложения Delphi

Примечание

Подобно обычному приложению БД, в "тонком" клиенте для размещения невизуальных компонентов доступа к данным необходимо использовать модули данных.

Для получения набора данных сервера компонент `TClientDataSet` взаимодействует с компонентом `TDataSetProvider` с помощью методов интерфейса `IProviderSupport` *(см. главу 28)*.

По существу, все уникальные функции клиентского приложения сосредоточены в компоненте `TClientDataSet`, изучением которого мы и займемся далее в этой главе. В остальном клиентское приложение не отличается от обычного приложения БД и при его разработке могут применяться стандартные методы.

Компонент *TClientDataSet*

Компонент `TClientDataSet` содержится в клиентской части многозвенного распределенного приложения. Он инкапсулирует набор данных, переданный при помощи компонента-провайдера из удаленного набора данных.

Компонент обеспечивает выполнение следующих основных функций:

- получение данных от удаленного сервера и передача ему сделанных изменений с использованием удаленного компонента-провайдера;
- представление набора данных при помощи локального буфера и поддержка основных операций, унаследованных от класса `TDataSet`;
- объединение записей набора данных при помощи агрегатных функций для получения суммарных данных;
- локальное сохранение набора данных в файле и последующее восстановление набора данных из файла;
- представление набора данных в формате XML.

Предком компонента `TClientDataSet` является класс `TDataSet`, поэтому `TClientDataSet` обладает таким же набором функций, что и обычный компонент, инкапсулирующий набор данных. Основное же отличие заключается в том, источник данных для него доступен только через удаленный компонент-провайдер. Это означает, что сохранение изменений и обновление набора данных осуществляется локально, без обращения к источнику данных.

Например, выполнение метода `Post` приведет лишь к сохранению текущей записи набора данных в локальном кэше. Все изменения отсылаются на сервер только при необходимости, и легко управляются разработчиком.

Как и обычный компонент, компонент `TClientDataSet` может употребляться совместно визуальными компонентами отображения данных. Для этого нужен компонент `TDataSource`.

Рассмотрим основные функции, реализуемые компонентом `TClientDataSet`.

Получение данных от компонента-провайдера

Компонент `TClientDataSet` получает доступ к удаленным данным через компонент соединения DataSnap `TDCOMConnection` *(см. главу 27)*. В зависимости от выбранной технологии это могут быть компоненты `TDCOMConnection`, `TSocketConnection` и `TWebConnection`.

Компонент `TClientDataSet` связывается с компонентом соединения при помощи свойства

```
property RemoteServer: TCustomRemoteServer;
```

Если соединение настроено правильно, то ссылка на интерфейс `IAppServer` в свойстве

```
property AppServer: IAppServer;
```

совпадает со свойством

```
ClientDataSet.RemoteServer.AppServer
```

После настройки соединения в свойстве

```
property ProviderName: string;
```

можно выбрать один из компонентов-провайдеров, которые доступны на сервере приложений, выбранном в компоненте соединения.

Если провайдер был подключен правильно, свойство только для чтения

```
property HasAppServer: Boolean;
```

автоматически принимает значение `True`.

Теперь компонент готов к приему данных. В случае метода

```
procedure Open;
```

или свойства

```
property Active: Boolean;
```

компонент получает от провайдера первый пакет данных.

Размер пакета определяется свойством

```
property PacketRecords: Integer;
```

которое задает число записей, передаваемое в одном пакете. Если свойство имеет значение −1 (это значение по умолчанию), передаются все записи набора данных. Если оно равно 0, клиенту передаются только метаданные о наборе данных.

Если соединение клиента с сервером медленное, число записей в пакете можно уменьшить, но желательно так, чтобы при использовании компонентов `TDBGrid` полученные в одном пакете записи полностью заполняли рабочую область этого компонента.

При этом разработчик имеет возможность управлять доставкой следующих пакетов. Для этого предусмотрен метод

```
function GetNextPacket: Integer;
```

Например, это можно сделать следующим образом:

```
procedure TDataModule1.ClientDataSetAfterScroll(DataSet: TDataSet);
begin
 if ClientDataSet.EOF then ClientDataSet.GetNextPacket;
end;
```

А свойство

```
property FetchOnDemand: Boolean;
```

должно иметь значение `False`. При значении `True` оно разрешает компоненту получать новые пакеты данных по мере необходимости, например, при необходимости прокрутки записей в компоненте `TDBGrid`.

До и после получения очередного пакета соответственно выполняются обработчики событий:

```
type
  TRemoteEvent = procedure(Sender: TObject; var OwnerData: OleVariant) of
object;
property BeforeGetRecords: TRemoteEvent;
property AfterGetRecords: TRemoteEvent;
```

Содержимое очередного пакета представлено свойством

```
property Data: OleVariant;
```

Данные в нем хранятся в транспортном формате, готовые для пересылки. Причем он доступен не только для чтения, но и для записи в виде пакета данных для отправки провайдеру:

```
var  OwnerData: OleVariant;
     MaxErrors, ErrorCount: Integer;
...
MaxErrors := 0;
  ResultDataSet.Data := SourceDataSet.AppServer.AS_ApplyUpdates
('', SourceDataSet.Delta, MaxErrors, ErrorCount, OwnerData);
...
```

Метод `AS_ApplyUpdates` передает данные, содержащиеся в буфере `Delta`, провайдеру на сервер и возвращает записи, сохранить которые не удалось. Подробнее о методе `AS_ApplyUpdates` см. табл. 28.1.

Размер буфера `Data` в байтах возвращает свойство

```
property DataSize: Integer;
```

Кэширование и редактирование данных

После получения записей от провайдера набор данных сохраняется в локальном буфере памяти. И все вносимые изменения после применения метода `Post` также сохраняются локально и не пересылаются на сервер. Буфер изменений доступен при помощи свойства

```
property Delta: OleVariant;
```

Для передачи изменений на сервер имеется метод

```
function ApplyUpdates(MaxErrors: Integer); Integer; virtual;
```

где параметр MaxErrors задает число ошибок, которые игнорируются при сохранении данных на сервере. Если параметр равен −1, сохранение на сервере прерывается при первой же ошибке. Метод возвращает число сохраненный записей.

После выполнения метода ApplyUpdates все записи, сохранить которые не удалось, возвращаются клиенту в локальный буфер Delta.

Если клиентское приложение будет редко изменять свои наборы данных, сохранение изменений на сервере можно связать с методом-обработчиком AfterPost:

```
procedure TForm1.ClientDataSetAfterPost(DataSet: TDataSet);
begin
 ClientDataSet.ApplyUpdates(-1);
end;
```

Свойство только для чтения

```
property ChangeCount: Integer;
```

возвращает общее число изменений, содержащееся в буфере Delta.

Для очистки буфера изменений есть метод

```
procedure CancelUpdates;
```

После вызова метода свойство ChangeCount принимает значение 0.

До и после сохранения изменений на сервере соответственно вызываются методы-обработчики

```
property BeforeApplyUpdates: TRemoteEvent;
property AfterApplyUpdates: TRemoteEvent;
```

Несмотря на сделанные локально многократные изменения, запись может быть восстановлена в первоначальном виде. Метод

```
procedure RefreshRecord;
```

получает от провайдера первоначальный вариант текущей записи, сохраненный на сервере. При этом (и при всех других случаях, когда компонент запрашивает обновление текущей записи) вызываются методы-обработчики

```
property BeforeRowRequest: TRemoteEvent;
property AfterRowRequest: TRemoteEvent;
```

Но что делать, если необходимо восстановить удаленную запись? В обычном наборе данных после сохранения это сделать невозможно. В компоненте TClientDataSet существует метод

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

который возвращает набор данных к состоянию до последней выполненной операции редактирования, добавления или удаления записи. Если параметр `FollowChange` имеет значение `True`, курсор набора данных будет установлен на восстановленную запись.

О состоянии текущей записи позволяет судить метод

```
function UpdateStatus: TUpdateStatus; override;
```

который возвращает значение типа

```
TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

означающее состояние текущей записи:

- `usUnmodified` — запись осталась неизменной;
- `usModified` — запись была изменена;
- `usInserted` — запись была добавлена;
- `usDeleted` — запись была удалена.

Например, при закрытии набора данных можно выполнить проверку:

```
if ClientDataSet.UpdateStatus = usModified
 then ShowMessage('Record was changed');
```

С помощью типа `TUpdateStatus` можно управлять видимостью записей в наборе данных.

Свойство

```
property StatusFilter: TUpdateStatusSet;
```

определяет, какой тип записей будет отображаться в наборе данных. Например,

```
ClientDataSet.StatusFilter := usDeleted;
```

отобразит в наборе данных только удаленные записи (при этом изменения не сохранены на сервере).

Управление запросом на сервере

Компонент `TClientDataSet` может не только эффективно управлять своим набором данных, но и влиять на выполнение серверного компонента, с которым он связан через провайдер.

Свойство

```
property CommandText: string;
```

содержит текст запроса SQL, имя таблицы или хранимой процедуры в зависимости от типа серверного компонента.

Изменив значение этого свойства на клиенте, можно, например, модифицировать запрос SQL на сервере. Но для этого в свойстве `Options` соответст-

вующего компонента-провайдера `TDataSetProvider` должно быть установлено значение

```
poAllowCommandText := True;
```

Новое значение свойства `CommandText` отправляется на сервер только после открытия клиентского набора данных или выполнения метода

```
procedure Execute; virtual;
```

Для запросов или хранимых процедур можно задавать параметры, которые сохраняются в свойстве

```
property Params: TParams;
```

До выполнения запроса присваиваются значения входным параметрам. После выполнения хранимой процедуры в выходных параметрах размещаются полученные от сервера значения.

Обратите внимание, что при выполнении запросов или хранимых процедур может измениться порядок следования параметров. Поэтому обращаться к параметрам желательно по их именам. Например, так:

```
Edit1.Text := ClientDataSet.Params.ParamByName('OutputParam').AsString;
```

Для того чтобы получить текущие значения параметров компонента набора данных на сервере, достаточно вызвать метод

```
procedure FetchParams;
```

До и после получения параметров от провайдера клиентский набор данных вызывает методы-обработчики событий:

```
property BeforeGetParams: TRemoteEvent;
property AfterGetParams: TRemoteEvent;
```

Использование индексов

Обычно работа с индексами — прерогатива сервера БД. Из компонентов Delphi в только табличные компоненты могут в какой-то степени управлять использованием индексов. Очевидно, что удаленное соединение не способствует эффективному управлению индексами набора данных на сервере. Поэтому компонент `TClientDataSet` предоставляет разработчику возможность реализовать локальные индексы.

Правильно созданные и используемые локальные индексы могут существенно ускорить выполнение операций с набором данных. В то же время их невозможно сохранить вместе с набором данных локально, их необходимо перестраивать при каждом новом открытии набора данных и его обновлении с сервера.

Для создания локального индекса служит метод

```
procedure AddIndex(const Name, Fields: string; Options: TIndexOptions;
const DescFields: string = ''; const CaseInsFields: string = ''; const
GroupingLevel: Integer = 0 );
```

Параметр `Name` определяет новое имя индекса. Параметр `Fields` должен содержать имена полей, которые разработчик хочет включить в индекс. Имена полей должны разделяться точкой с запятой. Параметр `Options` позволяет задать тип индекса:

```
TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive,
ixExpression, ixNonMaintained);
  TIndexOptions = set of TIndexOption;
```

Здесь:

- `ixPrimary` — первичный индекс;
- `ixUnique` — значения индекса уникальны;
- `ixDescending` — индекс сортирует записи в обратном порядке;
- `ixCaseInsensitive` — индекс сортирует записи без учета регистра символов;
- `ixExpression` — в индексе используется выражение (для индексов dBASE);
- `ixNonMaintained` — индекс не обновляется при открытии таблицы.

При этом можно задать поля, порядок сортировки которых будет обратным. Для этого их необходимо перечислить через точку с запятой в параметре `DescFields`. А параметр `CaseInsFields` аналогичным образом позволяет задать поля, на сортировку которых не влияет регистр символов.

Параметры `DescFields` и `CaseInsFields` заменяют параметр `Options`.

Параметр `GroupingLevel` задает уровень группировки полей индекса. Подробнее об этом см. далее.

Основные свойства компонента, обеспечивающие управление индексами, совпадают с аналогичными свойствами табличных компонентов (подробнее об этом *см. главу 26*). Поэтому лишь кратко перечислим их.

При работе с компонентом разработчик имеет возможность управлять индексами.

Созданный индекс подключается к набору данных свойством

```
property IndexName: String;
```

которое должно включать имя индекса, или свойство

```
property IndexFieldNames: String;
```

где можно задать произвольное сочетание имен индексированных полей таблицы. Имена полей разделяются точкой с запятой. Свойства `IndexName` и `IndexFieldNames` одновременно недопустимы.

Число полей в текущем индексе табличного компонента возвращает свойство

```
property IndexFieldCount: Integer;
```

А свойство

```
property IndexFields: [Index: Integer]: TField;
```

представляет собой индексированный список полей, входящих в текущий индекс.

Параметры созданных индексов доступны в свойстве

```
property IndexDefs: TIndexDefs;
```

Класс `TIndexDefs` подробно рассматривался в *главе 26*.

После создания и подключения индекса записи набора данных переупорядочиваются в соответствии со значениями индексированных полей.

Удаление локального индекса обеспечивает метод

```
procedure DeleteIndex(const Name: string);
```

После удаления текущего индекса или его отмены (обнуления свойства `IndexName`) записи набора данных переупорядочиваются в исходном порядке, соответствующем порядку записей набора данных на сервере.

Имена всех существующих в наборе данных индексов можно загрузить в список при помощи метода

```
procedure GetIndexNames(List: TStrings);
```

Например:

```
Memo1.Lines.Clear;
ClientDataSet.GetIndexNames(Memo1.Lines);
```

Сохранение набора данных в файлах

Клиентское приложение может использовать одну очень удобную функцию компонента `TClientDataSet`. Представим, что соединение между сервером и клиентом обладает малой пропускной способностью, и к тому же часто обрывается. Что в этом случае делать пользователю, который внес много изменений и не может сохранить их на сервере?

В этом случае можно сохранить набор данных клиента в файле на локальном диске, а при удобной возможности — загрузить обратно и переслать на сервер.

Для сохранения данных (по существу это буфер `Data`) в файле предусмотрен метод

```
procedure SaveToFile(const FileName: string = ''; Format:
TDataPacketFormat=dfBinary);
```

Причем если параметр `FileName` пуст, имя файла берется из свойства

```
property FileName: string;
```

Также можно передать данные в поток:

```
procedure SaveToStream(Stream: TStream; Format:
TDataPacketFormat=dfBinary);
```

Формат, в котором данные будут сохранены, определяется параметром `Format`:

```
type TDataPacketFormat = (dfBinary, dfXML, dfXMLUTF8);
```

где:

- `dfBinary` — бинарный;
- `dfXML` — формат XML;
- `dfXMLUTF8` — формат XML в кодировке UTF8.

Обратная загрузка данных, соответственно, выполняется методами:

```
procedure LoadFromFile(const FileName: string = '');
```

и

```
procedure LoadFromStream(Stream: TStream);
```

После загрузки набор данных полностью готов к работе:

```
if LoadFileDialog.Execute then
begin
 ClientDataSet.LoadFromFile(LoadFileDialog.FileName);
 ClientDataSet.Open;
end;
```

Работа с данными типа BLOB

Если набор данных сервера содержит большие поля (например, изображения), передача данных по медленному каналу займет очень много времени, что, несомненно, снизит эффективность приложения. Простейшее решение проблемы — передача клиенту данных типа BLOB только в том случае когда, это ему действительно необходимо, т. е. исключительно по его запросу.

В компоненте `TClientDataSet` процессом передачи полей типа BLOB можно управлять посредством свойства

```
property FetchOnDemand: Boolean;
```

По умолчанию оно равно `True` и клиентский набор данных "выкачивает" данные BLOB по мере необходимости автоматически. А это означает, что

приложение будет останавливаться и заново получать данные при любом просмотре данных, прокрутке и т. д. Если свойство имеет значение `False`, для получения данных клиент должен явно вызвать метод

```
procedure FetchBlobs;
```

Но кроме этого, в свойстве `Options` компонента-провайдера `TDataSetProvider` обязательно должно быть установлено значение:

```
poFetchBlobsOnDemand := True;
```

Представление данных в формате XML

Набор данных клиента легко можно представить в формате XML. Для этого достаточно применить свойство

```
property XMLData: OleVariant;
```

которое возвращает данные, содержащиеся в буфере `Data` в бинарном виде, в формате XML.

Например, клиентский набор данных можно сохранить в файле формата XML:

```
if SaveDialog.Execute then
 with TFileStream.Create(SaveDialog.FileName, fmCreate) do
 try
Write(Pointer(ClientDataSet.XMLData)^, Length(ClientDataSet.XMLData));
 finally
  Free;
 end;
```

Агрегаты

Наличие локального буфера данных позволяет компоненту `TClientDataSet` реализовать ряд дополнительных функций, основанных на агрегатных функциях применительно к полям всего набора данных, загруженного в локальный буфер. К агрегатным функциям относятся следующие:

- `AVG` — вычисляет среднее значение;
- `COUNT` — возвращает число записей;
- `MIN` — вычисляет минимальное значение;
- `MAX` — вычисляет максимальное значение;
- `SUM` — вычисляет сумму.

Для их применения в компоненте `TClientDataSet` предусмотрены конструкции.

- Индексированный список объектов, инкапсулирующих агрегатные выражения — агрегаты.
- Агрегатные поля, обеспечивающие получение новых значений подобно вычисляемым полям, но с группированием записей на основе агрегатных функций.

Объекты-агрегаты

Для вычисления агрегатных выражений для всех записей набора данных реализованы объекты класса `TAggregate`. Индексированный список этих объектов содержится в свойстве

```
property Aggregates: TAggregates;
```

компонента `TClientDataSet`. Прямым предком класса `TAggregates` является класс `TCollection`, поэтому для него пригодны все основные приемы работы с коллекциями.

Для создания нового агрегата необходимо щелкнуть на кнопке свойства в Инспекторе объектов и в появившемся Редакторе агрегатов выбрать пункт **Add** в всплывающем меню или щелкнуть на кнопке **Add New** (рис. 29.2).

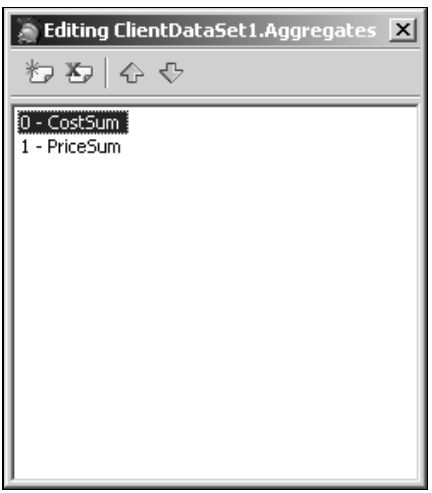


Рис. 29.2. Редактор агрегатов компонента `TClientDataSet`

Новый агрегат может быть добавлен и динамически:

```
var NewAgg: TAggregate;
...
NewAgg := ClientDataSet.Aggregates.Add;
...
```

Рассмотрим свойства класса `TAggregate`.

Имя агрегата содержится в свойстве

```
property AggregateName: string;
```

которое может быть использовано при отображении агрегата в визуальных компонентах.

Вычисляемое выражение с применением агрегатных функций должно находиться в свойстве

```
property Expression: String;
```

Например, для таблицы COUNTRY.DB из демонстрационной базы данных Delphi можно вычислять общую площадь государств Северной и Южной Америки (площадь государства содержится в поле `Area`):

```
ClientDataSet.Aggregates[SomeIndex].Expression := 'SUM(Area)';
```

Вычислением агрегата управляет свойство

```
property Active: Boolean;
```

а вычисленное значение возвращает функция

```
function Value: Variant;
```

Если пользователь редактирует набор данных, то для всех включенных агрегатов (`Active = True`) возвращаемое значение автоматически пересчитывается.

Например, после сохранения изменений в наборе данных можно визуализировать новое значение агрегата:

```
SomeLabel.Caption := ClientDataSet.Aggregates[0].AggregateName;
SomeEdit.Text := ClientDataSet.Aggregates[0].Value;
```

Для проверки активности агрегата помимо проверки значения свойства `Active` можно также вызвать свойство

```
property InUse: Boolean;
```

Если оно возвращает `True` — вычисляемое выражение агрегата рассчитывается.

Видимость агрегата в визуальных компонентах управляется свойством

```
property Visible: Boolean;
```

Для того чтобы снизить вычислительную нагрузку на набор данных, можно отключить все агрегаты одновременно. Для этого свойству

```
property AggregatesActive: Boolean;
```

необходимо присвоить значение `False`. Если же `AggregatesActive = True`, вычисляются только активные агрегаты, для которых их свойство `Active` имеет значение `True`.

Если вам необходимы все активные агрегаты, то вместо их последовательного перебора с проверкой свойства `Active` можно применить свойство

```
property ActiveAggs[Index: Integer] : TList;
```

компонента `TClientDataSet`, которое представляет собой список активных агрегатов.

Агрегатные поля

Агрегатные поля не входят в структуру полей набора данных, т. к. агрегатные функции подразумевают объединение записей таблицы для получения результата. Следовательно, значение агрегатного поля нельзя связать с какой-то одной записью, оно относится ко всем или группе записей.

Создать агрегатное поле можно командой **New field** из всплывающего меню Редактора полей компонента `TClientDataSet` (рис. 29.3).

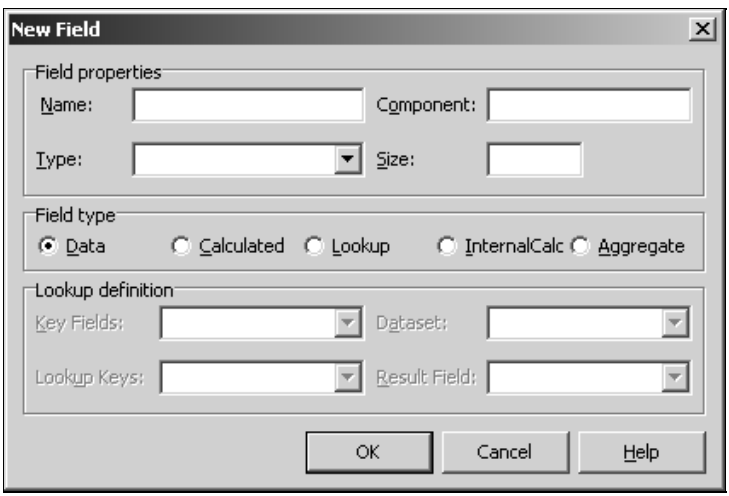


Рис. 29.3. Редактор полей компонента `TClientDataSet`

После выбора радиокнопки **Aggregate** в группе **Field Type** в списке выбора **Type** автоматически появляется тип поля `TAggregate`. Имя нового поля задается в однострочном редакторе **Name**.

Агрегатные поля не отображаются вместе со всеми полями в компонентах `TDBGrid`, а расположены в отдельном списке в нижней части окна Редактора полей (рис. 29.4).

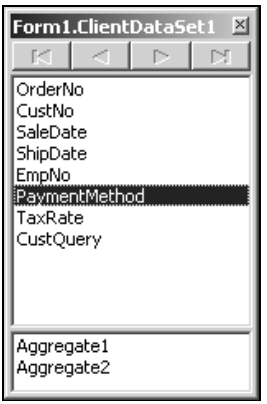


Рис. 29.4. Редактор полей
с отдельным списком агрегатных полей

Для представления значения агрегатного поля можно воспользоваться одним из компонентов отображения данных, который визуализирует значение одного поля (например, `TDBText` или `TDBEdit`) или свойствами самого поля:

```
Label1.Caption := MyDataSetAGGRFIELD1.AsString;
```

Класс `TAggregateField` предназначен для инкапсуляции свойств и методов агрегатных полей.

Его свойство

```
property Expression: string;
```

задает вычисляемое выражение.

Вычисление значения проводится только для тех агрегатных полей, свойство

```
property Active: Boolean;
```

которых имеет значение `True`.

Вычисление включенных свойством `Active` агрегатных полей выполняется только в том случае, если булево свойство `AggregatesActive` клиентского компонента набора данных имеет значение `True`.

По умолчанию экземпляр класса `TAggregateField` создается со свойством `Visible = False`.

Группировка и использование индексов

Каждый агрегат (объект или поле) имеет свойство

```
property GroupingLevel: Integer;
```

которое задает уровень группировки полей набора данных при вычислении. При значении 0 расчет проводится для всех записей набора данных. При значении 1 записи группируются по первому полю набора данных и расчет осуществляется для каждой группы. При значении 2 записи разбиваются на группы по первому и второму полям и т. д.

Однако группировка по уровням выше нулевого возможна, только если в наборе данных используется индекс по группирующим полям. Например, если свойство `GroupingLevel = 2` и набор данных начинается с полей `CustNo` и `OrderNo`, в свойстве `IndexName` компонента `TClientDataSet` и свойстве

```
property IndexName: String;
```

агрегата (объекта или поля) должно быть имя индекса, включающего оба эти поля.

Вложенные наборы данных

При организации между таблицами отношения "один ко многим" через одинаковое значение поля (внешнего ключа) одна запись главной таблицы связывается с несколькими записями подчиненной *(см. главу 15)*. Этот широко распространенный в практике программирования приложений БД механизм реализован и в компоненте `TClientDataSet`. Для этого есть класс поля `TDataSetField`.

На стороне клиента для создания отношения "один ко многим" необходимы как минимум два компонента `TClientDataSet`, главный из которых инкапсулирует основной набор данных, а подчиненный — вложенный набор данных.

Итак, на стороне сервера есть два табличных компонента, связанных отношением "один ко многим" при помощи свойств `MasterSource` и `MasterFields`. Также это могут быть и два компонента запросов SQL, связанные параметрами подчиненного запроса, с одноименными полями главного запроса и свойством `DataSource`.

Теперь на стороне клиента необходимо при помощи компонента-провайдера связать компонент `TClientDataSet` с главным серверным компонентом отношения "один ко многим" и создать для него статические объекты для всех полей. Для этого достаточно дважды щелкнуть на компоненте и в окне Редактора полей (см. рис. 29.3) из всплывающего меню выбрать пункт **Add Field**.

В результате в окне Редактора полей появятся имена объектов для всех полей данных серверного набора данных, а также еще одно дополнительное поле объектного типа `TDataSetField`. Его имя совпадает с именем подчиненного серверного компонента отношения "один ко многим".

Это поле связано с подчиненным компонентом на сервере. Чтобы убедиться в этом, достаточно просмотреть значение его свойства только для чтения

```
property NestedDataSet: TDataSet;
```

А индексированный список всех полей, передаваемых из серверного подчиненного компонента, содержится в свойстве только для чтения

```
property Fields: TFields;
```

В дальнейшем связь между компонентами на клиенте настраивается именно через это поле. В подчиненном компоненте `TClientDataSet` в Инспекторе объектов необходимо выбрать свойство

```
property DataSetField: TDataSetField;
```

В списке этого свойства вы увидите имя только что созданного поля объектного типа `TDataSetField`. Выберите его — и отношение "один ко многим" для клиентских наборов данных готово. При этом в компоненте вложенного набора данных автоматически очищаются свойства `RemoteServer` и `ProviderName`, т. к. их значения утрачивают значение и компонент оказывается связан только с главным компонентом отношения "один ко многим".

Теперь при навигации по записям основного набора данных во вложенном наборе данных автоматически будут появляться связанные записи. Также вы можете использовать все возможности, предоставляемые компонентом `TClientDataSet` как для основного, так и для вложенного набора данных.

Дополнительные свойства полей клиентского набора данных

Как известно, все классы полей имеют одного общего предка — класс `TField`. Здесь же остановимся лишь на нескольких дополнительных свойствах полей, которые работают только в режиме кэширования в обычных компонентах, инкапсулирующих набор данных, и в компоненте `TClientDataSet`. Причем в компоненте `TClientDataSet` реализация этих свойств обеспечена локальным кэшем.

Итак, для разработчика могут быть полезны свойства объектов полей, содержащие не только текущее, но и предыдущее значение поля.

Свойство

```
property CurValue: Variant;
```

возвращает текущее значение поля.

Свойство

```
property OldValue: Variant;
```

содержит значение поля, которое было до начала редактирования.

Свойство

```
property NewValue: Variant;
```

содержит новое значение, которое может быть присвоено при обработке ошибки сервера методом-обработчиком `OnReconcileError`.

Обработка ошибок

Особенности компонента `TClientDataSet` распространяются также и на обработку ошибок. Ведь клиентский набор данных должен реагировать не только на ошибки, возникшие локально, но и на ошибки сохранения изменений на сервере.

В первом случае разработчик может применить стандартные способы. Это блоки `try ... except` или методы-обработчики, унаследованные от класса `TDataSet`:

- `property OnDeleteError: TDataSetErrorEvent;` — вызывается при ошибках удаления записей;
- `property OnEditError: TDataSetErrorEvent;` — вызывается при ошибках редактирования записей;
- `property OnPostError: TDataSetErrorEvent;` — вызывается при ошибках локального сохранения записей.

Все они используют процедурный тип

```
type
  TDataSetErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError;
var Action: TDataAction) of object;
```

Здесь, помимо параметров `DataSet` и `E`, определяющих соответственно набор данных и тип ошибки, параметром `Action` можно задать вариант реакции на ошибку:

```
type TDataAction = (daFail, daAbort, daRetry);
```

Здесь:

- `daFail` — прервать операцию и показать сообщение об ошибке;
- `daAbort` — прервать операцию без сообщения об ошибке;
- `daRetry` — повторить операцию.

Например, при возникновении ошибки редактирования набора данных код обработчика может выглядеть следующим образом (листинг 29.1).

Листинг 29.1. Пример обработчика ошибки

```
procedure TForm1.ClientDataSetEditError(DataSet: TDataSet; E:
EDatabaseError; var Action: TDataAction);
begin
 if Not (DataSet.State in [dsEdit, dsInsert]) then
 begin
  DataSet.Edit;
  Action := daRetry;
 end
 else Action := daAbort;
end;
```

Здесь, если набор данных не находится в состоянии редактирования, это упущение исправляется и операция повторяется.

Итак, с локальными ошибками все обстоит достаточно просто. А как клиентский набор данных "узнает" об ошибке на удаленном сервере? Очевидно, при помощи своего компонента-провайдера. Действительно, компонент `TDataSetProvider` не только возвращает клиенту несохраненные изменения в пакете `Delta`, но и обеспечивает генерацию события, реакцией на которое является метод-обработчик

```
type
  TReconcileErrorEvent = procedure(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction)
of object;
property OnReconcileError: TReconcileErrorEvent;
```

Обратите внимание, что все параметры похожи на соответствующие параметры локальных обработчиков, но имеют собственные типы. Рассмотрим их.

Параметр `UpdateKind` содержит указание на тип операции, вызвавшей ошибку на сервере:

```
type
  TUpdateKind = (ukModify, ukInsert, ukDelete);
```

Здесь:

- `ukModify` — изменение данных;
- `ukInsert` — добавление записей;
- `ukDelete` — удаление записей.

Параметр `Action` позволяет разработчику предусмотреть реакцию клиентского набора данных на ошибку:

```
type
  TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel,
raRefresh);
```

Здесь:

- `raSkip` — отменить операцию для записей, вызвавших ошибку, с их сохранением в буфере;
- `raAbort` — отменить все изменения для операции, вызвавшей ошибку;
- `raMerge` — совместить измененные записи с аналогичными записями сервера;
- `raCorrect` — сохранить изменения, сделанные в данном методе-обработчике;
- `raCancel` — отменить изменения, вызвавшие ошибку, заменив их исходными локальными значениями клиентского набора данных;
- `raRefresh` — отменить изменения, вызвавшие ошибку, заменив их исходными значениями серверного набора данных.

Как видите, выбор возможных реакций на ошибку сервера несколько шире, чем на локальные ошибки.

Тип ошибки возвращается параметром `E`, для которого предусмотрен специальный класс `EReconcileError`, который имеет несколько полезных свойств.

Свойство

```
property ErrorCode: DBResult;
```

возвращает код ошибки. Коды ошибок можно найти в файле ..\Source\Win32\db\DSIntf.pas. Код предыдущей ошибки возвращается свойством

```
property PreviousError: DBResult;
```

Анализируя представленную здесь информацию, вы можете самостоятельно управлять обработкой ошибок сервера на клиенте. Но можно поступить и более просто — использовать стандартный диалог обработки удаленных ошибок (рис. 29.5). Этот диалог можно подключить к вашему проекту (он содержится в модуле ..\ObjRepos\DelphiWin32\RecError.pas) и вызывать при помощи процедуры

```
function HandleReconcileError(DataSet: TDataSet; UpdateKind: TUpdateKind;
ReconcileError: EReconcileError): TReconcileAction;
```

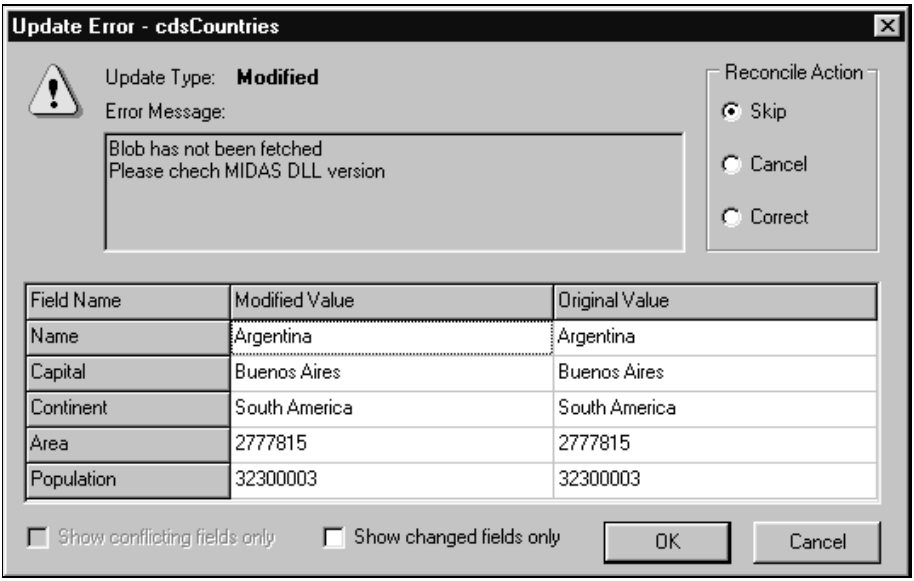


Рис. 29.5. Стандартный диалог обработки ошибок сервера

В параметры этой функции подставляются параметры метода-обработчика `OnReconcileError`, а возвращает она действие, выбранное пользователем в диалоге. Таким образом, ее использование очень просто:

```
procedure TForm1.ClientDataSetReconcileError(DataSet:
TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action:
TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Пример "тонкого" клиента

Пример клиентского приложения является частью группы проектов SimpleRemote.bdsgroup и предназначен для взаимодействия с сервером приложений `SimpleAppSrvr`, процесс создания которого подробно рассматривался в *главе 28*.

Проект клиента SimpleClient состоит из двух файлов.

- Компоненты, обеспечивающие соединение с удаленным сервером приложения и работу с наборами данных, сосредоточены в модуле данных

`DataModule` (файл uDataModule.pas). Обратите внимание, что это "обычный" модуль данных, используемый в приложениях баз данных *(см. главу 15)*.

- Главная форма клиентского приложения `fmMain` (файл uMain.pas), содержащая визуальные компоненты пользовательского интерфейса.

На рис. 29.6 показано окно клиентского приложения SimpleClient, в листинге 29.2 приведен код секции `implementation` модуля данных `DataModule`.

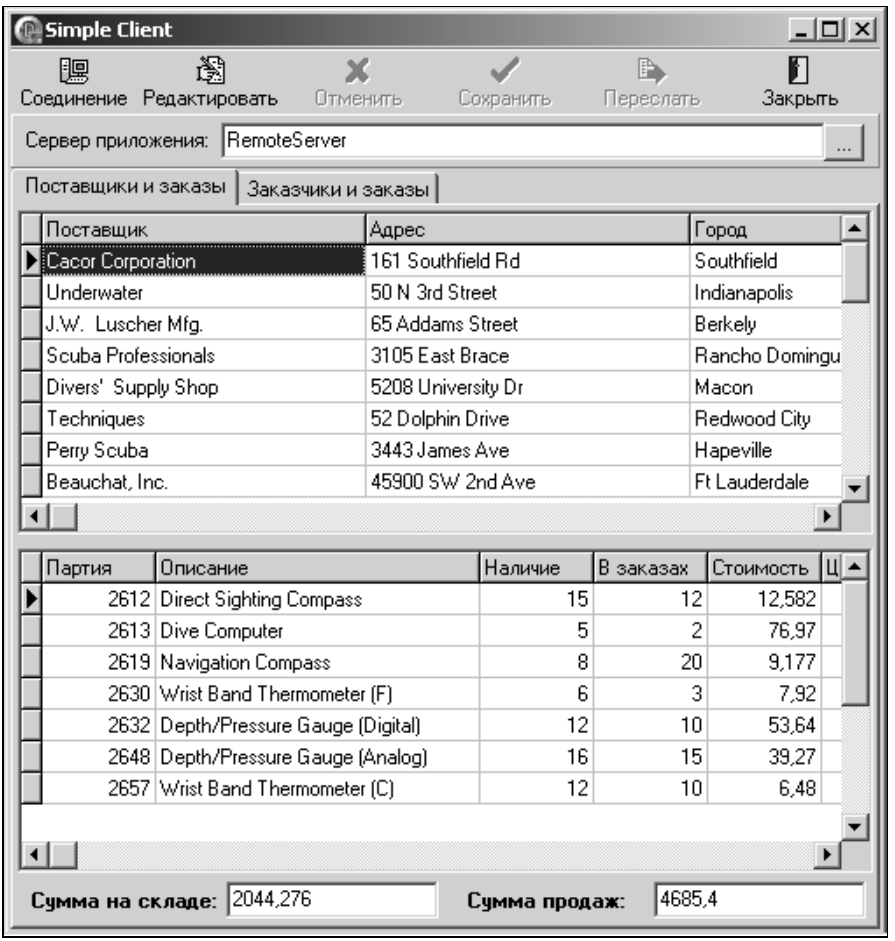


Рис. 29.6. Окно клиентского приложения SimpleClient

Листинг 29.2. Секция `implementation` модуля данных `DataModule`

```
implementation
uses uMain, Variants, Dialogs;
{$R *.dfm}
procedure TDM.SrvrConAfterConnect(Sender: TObject);
var i: Integer;
begin
 for i := 0 to SrvrCon.DataSetCount - 1 do
  SrvrCon.DataSets[i].Open;
 cdsVendors.Open;
end;
procedure TDM.SrvrConBeforeDisconnect(Sender: TObject);
var i: Integer;
begin
 for i := 0 to SrvrCon.DataSetCount - 1 do
  SrvrCon.DataSets[i].Close;
 cdsVendors.Close;
end;
procedure TDM.cdsVendorsAfterScroll(DataSet: TDataSet);
begin
 fmMain.edCostSum.Text := VarToStr(cdsParts.Aggregates[0].Value);
 fmMain.edPriceSum.Text := VarToStr(cdsParts.Aggregates[1].Value);
end;
procedure TDM.cdsPartsReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
 cdsParts.CancelUpdates;
 MessageDlg(E.Message, mtError, [mbOK], 0);
end;
end.
```

Соединение клиента с сервером приложения

Для соединения клиентского приложения с сервером в локальной сети использован компонент `SrvrCon` класса `TDCOMConnection`. Данный тип соединения выбран как наиболее простой и требующий лишь наличия локальной сети или даже не требующий ничего — в демонстрационном приложении

можно задействовать сервер приложения, установленный на этом же компьютере.

Для настройки соединения компонента `SrvrCon` в свойстве `ComputerName` было указано имя компьютера сервера. После этого в списке свойства `ServerName` можно выбрать один из доступных зарегистрированных серверов. В нашем случае это сервер `SimpleAppSrvr.SimpleRDM`, имя которого состоит из имени приложения сервера и имени главного удаленного модуля данных.

Обратите внимание, что в этом же списке имеется и дочерний модуль `Secondary`. Однако для получения доступа к наборам данных дочернего модуля данных мы не будем создавать еще одно соединение, а воспользуемся компонентом `TSharedConnection`, т. к. он специально предназначен для подобных случаев. Для его настройки достаточно указать в свойстве `ParentConnection` компонент соединения. В нашем случае это `SrvrCon`.

Для компонента `SrvrCon` предусмотрены два метода-обработчика (см. листинг 29.2): после подключения и перед отключением соединения. В них открываются и закрываются все наборы данных клиентского приложения.

Теперь в клиентском приложении доступны наборы данных обоих удаленных модулей данных сервера приложений.

Непосредственно подключение к серверу осуществляется кнопкой **Connect**. При ее нажатии выполняется следующий простой код (листинг 29.3).

Листинг 29.3. Код подключения к серверу

```
procedure TfmMain.tbConnectClick(Sender: TObject);
begin
 try
  DM.SrvrCon.Close;
  DM.SrvrCon.ComputerName := edServerName.Text;
  DM.SrvrCon.Open;
 except
  on E: Exception do MessageDlg(E.Message, mtError, [mbOK], 0);
 end;
 SetCtrlState;
end;
```

Соединение закрывается, задается новое имя компьютера сервера, соединение открывается. Специально созданный метод формы `SetCtrlState` управ-

ляет доступностью кнопок формы, анализируя текущее состояние наборов данных.

Наборы данных клиентского приложения

Каждый из компонентов `TClientDataSet` модуля данных `DataModule` связан с соответствующим компонентом-провайдером сервера.

Компонент `cdsOrders` предназначен для просмотра данных о заказах. Вспомогательные компоненты `cdsEmployees` и `cdsCustomers` содержат списки заказчиков и работников, используемые в главном наборе данных. В компоненте `cdsOrders` определено агрегатное поле `PaidSum`, рассчитывающее сумму платежей по всем заказам.

Компонент `cdsParts` предназначен для просмотра и редактирования данных о поступлениях. Компонент `cdsVendors` представляет список поставщиков. Так как в сервере приложения связанный с `cdsVendors` набор данных является главным в отношении "один ко многим", то одновременно с обычными полями для компонента `cdsVendors` автоматически создается поле `tblParts` типа `TDataSetField`. Это поле позволяет настроить вложенный набор данных. Для этого достаточно в свойстве `DataSetField` вложенного компонента `cdsParts` задать поле `tblParts`. Теперь при перемещении по записям главного набора данных `cdsVendors` вложенный набор данных компонента `cdsParts` будет отображать записи, связанные с текущим поставщиком.

Примечание

В целях сохранения простоты и наглядности исходного кода редактирование предусмотрено только для одного компонента `cdsParts`. В реальной работе аналогичные методы могут использоваться для всех наборов данных.

Для компонента `cdsParts` созданы два агрегата, суммирующие данные о поступлениях и продажах. При перемещении по записям этого набора данных в методе-обработчике `AfterScroll` предусмотрено обновление значений агрегатов (см. листинг 29.2).

Так как компонент `cdsParts` предназначен и для редактирования данных, то для него необходимо предусмотреть обработку исключительных ситуаций, возникающих не только на клиенте, но и на сервере. Для этого предусмотрен метод-обработчик `cdsPartsReconcileError` (см. листинг 29.2). Сама операция очень проста и скорее служит лишь демонстрацией возможности создавать собственную обработку серверных исключений вместо использования стандартной функции `HandleReconcileError` (см. рис. 29.5). Здесь все изменения в проблемных записях отменяются методом `CancelUpdates` и выводится сообщение об ошибке.

Локальное редактирование, сохранение или отмена изменений для компонента `cdsParts` выполняются стандартными методами набора данных. Дополнительно при отмене изменений вызывается метод `UndoLastChange`, позволяющий полностью восстановить последнюю модифицированную запись даже после локального сохранения изменений.

Передача изменений серверу реализована методом `ApllyUpdates`. Параметр −1 означает, что клиенту будет возвращено сообщение о первой же ошибке.

Резюме

В многозвенных распределенных приложениях в основном используются "тонкие" клиенты, делегирующие большинство функций ПО промежуточного слоя. В трехзвенных приложениях это сервер приложений.

Основой клиентского приложения является компонент `TClientDataSet`, который инкапсулирует набор данных и обеспечивает его функционирование при помощи локального буфера. Соединение с удаленным сервером приложений осуществляется через компоненты DataSnap.

Глава 30

Клиент распределенного приложения на основе данных XML

Как уже говорилось, Delphi предоставляет разработчикам возможность работы с данными в формате XML. Это возможно для любых типов распределенных приложений, в том числе приложений баз данных, Web-приложений, XML- и Web-служб.

В результате разработанные в Delphi приложения могут использовать в качестве источника данных не только традиционные базы данных, электронные таблицы, текстовые файлы, но и файлы XML. Кроме этого, отсюда всего один шаг до протокола SOAP и пересылки данных в формате XML.

Утилита преобразования данных XML Mapper позволяет преобразовывать данные на этапе разработки и описана в *главе 14*. Во время работы приложения всю необходимую работу выполнят компоненты преобразования данных из категории **Data Access** Палитры инструментов.

В этой главе рассматриваются следующие вопросы.

- Взаимодействие форматов XML и пакетов данных Delphi.
- Компонент преобразования данных XML — `TXMLTransform`.
- Компоненты доступа к данным XML.

Преобразование данных XML в распределенных приложениях

Теперь обратимся к программным аспектам работы с данными XML в распределенных приложениях. И в первую очередь рассмотрим вопрос преобразования данных XML. Для этого предназначен компонент `TXMLTransform`, который умеет преобразовывать данные из формата XML в пакет данных Delphi и обратно. Он является базовым, т. к. используется и в других компонентах, о которых речь пойдет далее.

В соответствии со схемой трансформации (см. рис. 14.1), необходимо определить источник и приемник данных, а также файл трансформации.

В качестве источника данных могут выступать как данные XML, так и пакет данных.

Свойство

```
property SourceXml: string;
```

позволяет задать исходные данные в виде строки, однако не спешите открывать файл в потоке и переносить из него данные. Для этого вполне достаточно указать в свойстве

```
property SourceXmlFile: string;
```

путь к файлу-источнику. Будучи задано, это свойство имеет приоритет перед другими свойствами, задающими исходный файл.

Кроме этого, можно открыть файл XML при помощи свойства

```
property SourceXmlDocument: IDOMDocument;
```

Оно должно содержать ссылку на компонент `TXMLDocument`, в котором и задается исходный файл. Но для того, чтобы это свойство сработало, необходимо, чтобы свойства `SourceXml` и `SourceXmlFile` были пусты.

Трансформационный файл задается свойством

```
property TransformationFile: string;
```

или, если схема трансформации определена документом XML в компоненте `TXMLDocument`, можно использовать свойство

```
property TransformationDocument: IDOMDocument;
```

После выполнения трансформации результат доступен в свойстве

```
property ResultString: string;
```

в формате XML.

После того как источник данных и схема трансформации определены, для преобразования данных достаточно обратиться к свойству

```
property Data: string;
```

При каждом обращении к нему преобразование выполняется автоматически. При обращении к этому свойству вызывается метод-обработчик

```
type
  TTranslateEvent = procedure(Sender: TObject; Id: string; SrcNode:
  IDOMNode; var Value: string; DestNode: IDOMNode) of object;
property OnTranslate: TTranslateEvent;
```

Например, с диалогом выбора файла XML и клиентским набором данных может быть связан следующий код (листинг 30.1).

Листинг 30.1. Пример преобразования данных XML

```
procedure TForm1.TransformClick(Sender: TObject);
const DefaultTranslationFile: String = 'CountryTrans.xtr';
begin
 OpenDlg.Filter := 'XML Files|*.xml';
 if OpenDlg.Execute then
 begin
  XMLTransform.SourceXmlFile := OpenDlg.FileName;
  XMLTransform.TransformationFile := DefaultTranslationFile;
  memo1.Lines.Add(XMLTransform.Data);
  ClientDataSet.Close;
  ClientDataSet.XMLData := XMLTransform.Data;
  ClientDataSet.Open;
 end;
end;
```

В этом примере выбирается файл XML и компонент `XMLTransform` при помощи трансформационного файла преобразует данные в пакет данных. Это происходит при вызове свойства `Data` и результат передается клиентскому набору данных `ClientDataSet` (использовано его свойство `XMLData`).

Кроме этого, вы можете воспользоваться методом

```
function TransformXML(const SourceXml: string; const ATransformationFile:
string = ''): string;
```

но для него необходимо задать файл-источник данных и файл трансформации заново при помощи параметров `SourceXml` и `ATransformationFile`.

Листинг 30.2 содержит фрагмент кода, соответствующий этому методу.

Листинг 30.2. Применение метода `TransformXML`

```
procedure TForm1.TransformClick(Sender: TObject);
const DefaultTranslationFile: String = 'CountryTrans.xtr';
var S: String;
begin
 OpenDlg.Filter := 'XML Files|*.xml';
 if OpenDlg.Execute then
 begin
  with TFileStream.Create(OpenDlg.FileName, fmOpenRead) do
```

```
  try
   SetString(S, nil, Size);
   Read(Pointer(S)^, Size);
  finally
   Free;
  end;
  XMLTransform.SourceXml := S;
  XMLTransform.TransformationFile := DefaultTranslationFile;
  ClientDataSet.Close;
  ClientDataSet.XMLData :=
XMLTransform.TransformXML(XMLTransform.SourceXml,
DefaultTranslationFile);
  ClientDataSet.Open;
 end;
end;
```

Результат преобразования может также сохраняться в документе XML, если определено свойство

```
property ResultDocument: IDOMDocument;
```

Дополнительно разработчик может заменить схему итогового документа, которая, как известно, хранится в файле трансформации. Для этого достаточно задать новую схему в свойстве

```
property EmptyDestinationDocument: IDOMDocument;
```

Схема XML должна быть загружена в компоненте `TXMLDocument`.

Кроме метода-обработчика `OnTranslate` компонент имеет две пары методов-обработчиков.

Одна пара

```
type
  TRowEvent = procedure(Sender: TObject; Id: string; SrcNode: IDOMNode;
  DestNode: IDOMNode) of object;
property BeforeEachRow: TRowEvent;
property AfterEachRow: TRowEvent;
```

вызывается до и после обработки одной записи.

Вторая пара

```
property BeforeEachRowSet: TRowEvent;
property AfterEachRowSet: TRowEvent;
```

вызывается до и после обработки набора записей.

Параметр `Id` содержит идентификатор (имя) преобразуемого элемента. Параметры `SrcNode` и `DestNode` являются ссылками на DOM-интерфейсы для исходного и целевого узлов соответственно.

Использование данных XML в распределенных приложениях

Для передачи данных XML клиенту распределенного приложения необходимо использовать компонент-провайдер, так же, как это делается и в распределенных приложениях баз данных *(см. главу 29)*. Естественно, что для работы с файлами XML провайдер должен быть специализированным. Это компонент `TXMLTransformProvider`.

Он является потомком класса `TCustomProvider` и предоставляет клиенту интерфейс `IAppServer`. Благодаря этому он выполняет основную функцию компонента-провайдера: передает данные от источника данных клиенту и принимает от клиента и передает источнику данных сделанные изменения. Источником данных для компонента являются файлы в формате XML. Для определения файла предназначено свойство

```
property XMLDataFile: string;
```

Для того чтобы клиентский набор данных (например, компонент `TClientDataSet`) мог работать с данными XML, их необходимо преобразовать в пакет данных Delphi. Обратная операция выполняется при передаче изменений назад.

Вы уже знаете, для таких преобразований необходимы трансформационные файлы. В случае с компонентом `TXMLTransformProvider` разработчики Borland поступили просто. Они интегрировали в компонент-провайдер два компонента `TXMLTransform`, которые и выполняют необходимые трансформации.

Операцию чтения данных их файла XML осуществляет компонент, доступный через свойство

```
property TransformRead: TXMLTransform;
```

Операцию записи изменений в файл XML выполняет компонент, доступный через свойство

```
property TransformWrite: TXMLTransform;
```

Для успешной работы каждому из этих компонентов необходимо придать трансформационный файл. Для этого есть их свойство `TransformationFile`.

Например:

```
XMLTransProvider1.TransformRead.TransformationFile := OpenDlg.FileName;
```

Кроме этого, для каждого из трансформационных компонентов могут быть определены их методы-обработчики.

А сам компонент `TXMLTransformProvider` обладает стандартным набором методов-обработчиков компонента-провайдера *(см. главу 29)*.

На основе компонента `TXMLTransformProvider` можно создавать распределенные приложения различных видов (рис. 30.1).

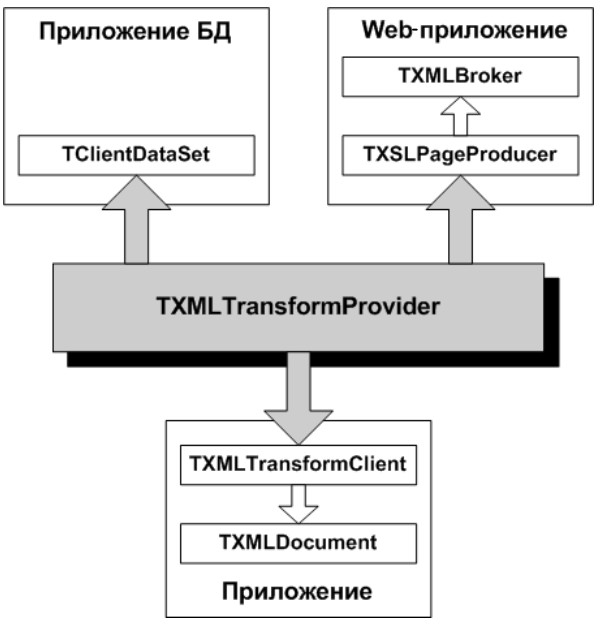


Рис. 30.1. Варианты использования компонента `TXMLTransformProvider` в распределенных приложениях

Во-первых, компонент можно сочетать с клиентскими наборами данных в распределенных приложениях баз данных. Он изолирует клиентское приложение от источника данных, и набор данных обращается через компонент-провайдер `TXMLTransformProvider` к файлам XML совершенно так же, как и к любой таблице в базе данных сервера.

Во-вторых, компонент позволяет передавать в клиентское приложение данные XML и работать с ними, как с документами XML. Для этого нужно применять компонент-провайдер совместно с компонентом `TXMLTransformClient`.

В-третьих, компонент будет полезен и в распределенных Web-приложениях. Для публикации данных XML необходимо компонент-провайдер связать со страницей HTML мостиком из компонента `TXMLBroker`.

Рассмотрим эти возможности более подробно.

Данные XML в клиентском наборе данных

Компонент `TXMLTransformProvider` имеет все необходимые свойства и методы стандартного компонента-провайдера. Поэтому он может взаимодействовать с компонентом `TClientDataSet` как локально, так и через удаленный сервер.

Подключение клиентского набора данных осуществляется, как и обычно, свойством `ProviderName`, а если источник данных XML расположен на удаленном сервере, то необходимо задать и сам сервер в свойстве `RemoteServer` *(см. главу 29)*.

В дальнейшем все операции по передаче данных XML клиенту и сохранению изменений выполняются стандартным образом.

Для локального изменения данных существуют методы редактирования набора данных (`Edit`, `Append`, `Post`, `Cancel` и т. д.), а для передачи изменений в исходный файл XML вызывается метод `ApplyUpdates` клиентского набора данных.

Данные XML в документе XML

Помимо традиционного способа, когда данные XML интегрированы в набор, клиентское приложение может работать с ними напрямую в формате документа XML. Для этого предназначен компонент `TXMLTransformClient`.

Он также может взаимодействовать с локальным или удаленным провайдером данных. Требуемый компонент-провайдер `TXMLTransformProvider` задается свойством

```
property ProviderName: string;
```

Если данные XML необходимы на удаленном сервере, нужно дополнительно задать свойство

```
property RemoteServer: TCustomRemoteServer;
```

Для обеспечения взаимодействия с источником данных XML компонент `TXMLTransformClient` инкапсулирует три компонента `TXMLTransform`. Они доступны через следующие свойства.

Свойство

```
property TransformGetData: TXMLTransform;
```

дает ссылку на компонент трансформации, обслуживающий входящий поток данных. В зависимости от формата этих данных его свойство `TransformationFile` должно указывать на файл трансформации, обеспечивающий правильное преобразование данных.

Для получения данных от провайдера и преобразования их в формат XML служит метод

```
function GetDataAsXml(const PublishTransformFile: string): string;
```

где параметр `PublishTransformFile` есть имя файла трансформации. Но если этот параметр пуст, имя файла будет взято из свойства `TransformGetData.TransformationFile`.

Например, так клиентское приложение может использовать документ XML, полученный от удаленного сервера:

```
...
SomeXMLDocument.XML.Clear;
SomeXMLDocument.XML.Add(XMLTransformClient.GetDataAsXml('SomwFilePath'));
...
```

Свойство

```
property TransformApplyUpdates: TXMLTransform;
```

обеспечивает доступ к компоненту трансформации, который преобразует исходящий поток модифицированных данных. В зависимости от формата этих данных его свойство `TransformationFile` должно указывать на файл трансформации, обеспечивающий правильное преобразование данных.

Если провайдер, связанный с компонентом `TXMLTransformClient`, обеспечивает передачу данных от запроса SQL с параметрами или хранимой процедуры, свойство

```
property TransformSetParams: TXMLTransform;
```

представляет собой компонент `TXMLTransform`, преобразующий параметры набора данных.

Если набор данных имеет параметры, то перед получением данных от провайдера и их преобразованием для компонента `TXMLTransformClient` необходимо вызвать метод

```
procedure SetParams(const ParamsXml, ParamsTransformFile: string);
```

который обеспечивыает преобразование параметров набора данных.

Параметр `ParamsTransformFile` указывает на файл трансформации, а параметр `ParamsXml` должен ссылаться на документ XML для преобразованных входящих параметров.

Данные XML на странице HTML

Компонент `TXMLTransformProvider` может также обеспечивать передачу и преобразование данных для Web-страниц. В этом случае "мостом" между компонентом-провайдером и компонентом-продюсером выступает компо-

нент TXMLBroker из категории **Internet Express** Палитры инструментов. Его задача — обеспечить "взаимопонимание" между Web-страницей, использующей преимущественно язык HTML, и провайдером TXMLTransformProvider, который работает с данными в формате XML.

Для соединения с провайдером компонент TXMLBroker имеет обычные свойства: ProviderName и RemoteServer. Для получения данных от сервера в распоряжении разработчика есть метод

```
function GetXMLRecords(var RecsOut: Integer; var OwnerData: OleVariant;
XMLOptions: TXMLOptions): string;
```

хотя обычно пересылка данных Web-странице и сохранение изменений выполняются автоматически.

Сам же компонент TXMLBroker должен взаимодействовать с компонентом TXSLPageProducer, который обеспечивает генерацию Web-страницы. Поэтому ссылка TXMLBroker должна быть определена в свойстве XMLData компонента-продюсера. Подробнее об этом способе публикации данных с точки зрения создания распределенного Web-приложения *см. главу 41.*

Для работы с Web-страницей компонент TXMLBroker имеет свойство

```
property WebDispatch: TWebDispatch;
```

которое представляет инкапсулированный компонент-диспетчер. С его помощью разработчик сможет получить исчерпывающую информацию о HTTP-запросах, подлежащих обработке.

Кроме этого, метод

```
function GetDelta(Request: TWebRequest): string;
```

возвращает дельта-пакет с изменениями, содержащимися в HTTP-запросе Request. А затем можно вызвать метод

```
function ApplyXMLUpdates(const Delta: string; out ErrorCount: Integer):
string;
```

который направит дельта-пакет компоненту-продюсеру. Например, при возникновении ошибки пересылки изменений в компоненте SomeXMLBroker запрос Web-страницы с изменениями может быть перенаправлен резервному компоненту TXMLBroker:

```
var DeltaStr: String;
    ErrCnt: Integer;
...
DeltaStr := SomeXMLBroker.GetDelta(SomeWrongRequest);
ReserveXMLBroker.ApplyXMLUpdates(DeltaStr, ErrCnt);
...
```

При пересылке данных от провайдера Web-странице и при возврате изменений обратно вызывается пара методов-обработчиков соответственно:

```
type TRequestRecordsEvent = procedure (Sender: TObject; Request:
TWebRequest; out RecCount: Integer; var OwnerData: OleVariant;
var Records: string) of object;
property OnRequestRecords: TRequestRecordsEvent;
```

и

```
type TRequestUpdateEvent = procedure (Sender: TObject; Request:
TWebRequest; Response: TWebResponse; var Handled: Boolean) of object;
property OnRequestUpdate: TRequestUpdateEvent;
```

Компонент `TXMLBroker` может регулировать размер пакета данных, передаваемого провайдером серверу. Для этого предназначено свойство

```
property MaxRecords: Integer;
```

которое задает число записей в одном пакете. По умолчанию свойство имеет значение −1, что означает, что размер пакета не регулируется. Если присвоить свойству значение 0, то записи вообще будут исключены из пересылаемых пакетов и в них будет содержаться лишь информация о структуре данных.

Компонент `TXMLBroker` имеет представительный набор свойств и методов для обработки возможных исключительных ситуаций.

Во-первых, он может вызвать компонент-продюсер, специально предназначенный для обработки ошибок, возникающих при пересылке данных компонентом-провайдером. Если свойство

```
property ReconcileProducer: TCustomContentProducer;
```

заполнено, соответствующий компонент-продюсер будет возвращать Web-странице ответы на запросы, вызвавшие исключительные ситуации.

Свойство

```
property MaxErrors: Integer;
```

позволяет задать число ошибок пересылки данных провайдером, которые игнорируются. Это свойство может быть полезно при плохом канале связи, когда для передачи пакета требуется несколько попыток.

После возникновения ошибки содержимое дельта-пакета можно получить при помощи метода

```
function GetErrors: string;
```

и общее число ошибок в этом пакете вернет метод

```
function GetErrorCount: Integer;
```

А сделать это проще всего методом-обработчиком

```
type TGetErrorResponseEvent = procedure (Sender: TObject; ErrorCount:
Integer; XMLErrors: string; Request: TWebRequest; Response: TWebResponse;
var Handled: Boolean) of object;
property OnGetErrorResponse: TGetErrorResponseEvent;
```

который вызывается при возникновении ошибки пересылки данных.

Пример приложения, использующего данные XML

В качестве примера создадим приложение, которое позволяет просматривать и редактировать данные XML в виде набора данных и документа XML (рис. 30.2). В качестве источника данных возьмем файл Country.xml из демонстрационной базы данных Delphi, но пользовательский интерфейс приложения позволяет выбрать и загрузить любой файл XML.

Компонент `TXMLTransformProvider` обеспечивает доступ к источнику данных. Для этого его в свойство `XMLDataFile` передается полный путь к файлу XML. Кроме этого, компонент-провайдер должен задействовать трансформационные файлы для чтения данных и сохранения сделанных изменений. Поэтому при открытии файла данных его свойствам

```
XMLProvider.TransformRead.TransformationFile := edReadFileName.Text;
```

и

```
XMLProvider.TransformWrite.TransformationFile := edWriteFileName.Text;
```

передаются имена соответствующих файлов.

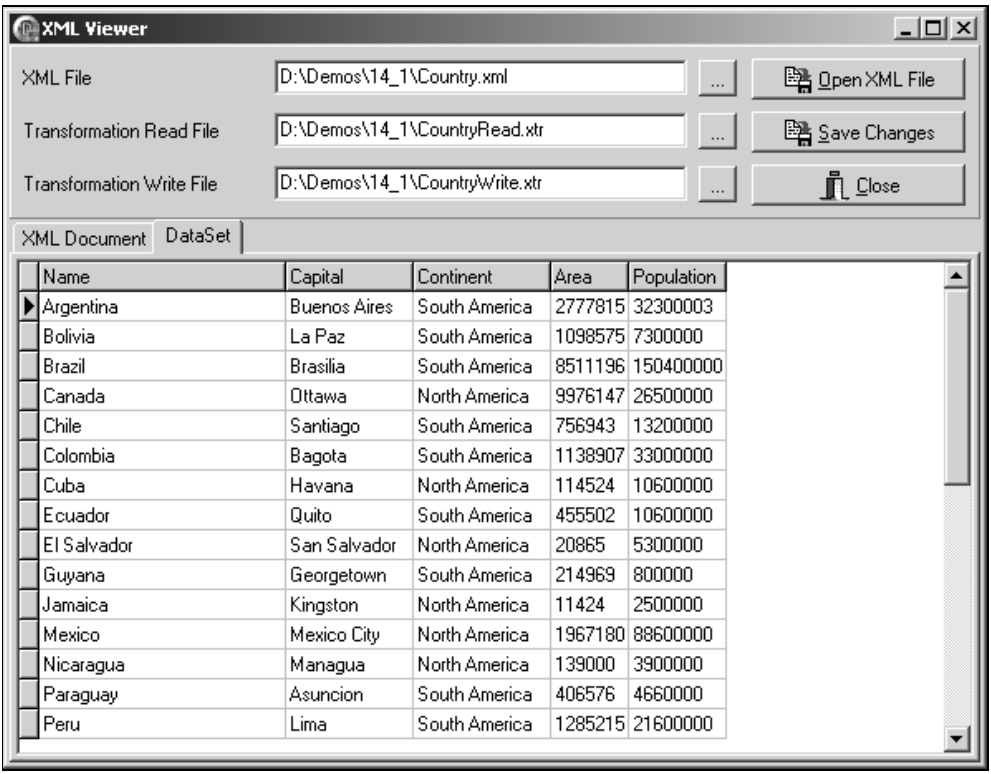
Теперь немного подробнее остановимся на создании трансформационных файлов. Для файла Country.xml они были созданы при помощи утилиты XML Mapper.

Сначала в нее был загружен исходный файл XML, затем для него был создан и сохранен файл пакета данных. После этого при помощи кнопки **Create and Test Transformation** создается и сохраняется файл трансформации для чтения данных XML CountryRead.xml. Затем при помощи группы переключателей **Transform Direction** меняется направление преобразования так, чтобы источником данных был файл пакета данных. После этого снова создается и сохраняется файл трансформации CountryWrite.xml, но уже для преобразования данных XML в пакет данных.

Для представления данных XML в наборе данных применен компонент `TClientDataSet`, который соединен с провайдером при помощи свойства `ProviderName`. Клиентский набор данных можно просмотреть и редактиро-

вать в компоненте `TDBGrid`. А сделанные изменения сохраняются через компонент-провайдер с помощью метода

```
ClDataSet.ApplyUpdates(-1);
```



Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32300003
Bolivia	La Paz	South America	1098575	7300000
Brazil	Brasilia	South America	8511196	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota	South America	1138907	33000000
Cuba	Havana	North America	114524	10600000
Ecuador	Quito	South America	455502	10600000
El Salvador	San Salvador	North America	20865	5300000
Guyana	Georgetown	South America	214969	800000
Jamaica	Kingston	North America	11424	2500000
Mexico	Mexico City	North America	1967180	88600000
Nicaragua	Managua	North America	139000	3900000
Paraguay	Asuncion	South America	406576	4660000
Peru	Lima	South America	1285215	21600000

Рис. 30.2. Главное окно приложения XML Viewer

Представление данных в документе XML реализуют компоненты `TXMLTransformClient` и `TXMLDocument`. Первый из них связан с провайдером свойством `ProviderName`. А второй заполняется данными при нажатии на кнопку `bbOpen`. Для этого вызывается метод `GetDataAsXml` компонента `TXMLTransformClient`:

```
XMLDoc.XML.Clear;
XMLDoc.XML.Add(XMLClient.GetDataAsXml(edReadFileName.Text));
```

Преобразование данных можно осуществить тем же трансформационным файлом, что и чтение данных XML компонента-провайдера.

Затем при помощи метода `SetXMLNode` заполняется дерево узлов и элементов XML в компоненте `TTreeView` (листинг 30.3).

Листинг 30.3. Метод `SetXMLNode` создает иерархическое дерево узлов и элементов документа XML при открытии файла XML

```
procedure TfmMain.SetXMLNode(TreeNode: TTreeNode; XMLNode: IXMLNode);
var i: Integer;
    NewNode: TTreeNode;
begin
 NewNode := tvXMLDoc.Items.AddChild(TreeNode, XMLNode.NodeName);
 if not Assigned(NewNode)
  then Abort;
 if Assigned(XMLNode.AttributeNodes) then
  for i := 0 to XMLNode.AttributeNodes.Count - 1  do
   tvXMLDoc.Items.AddChild(NewNode, XMLNode.AttributeNodes[i].NodeName +
              '='  +
               XMLNode.Attributes[XMLNode.AttributeNodes[i].NodeName]);
 if Assigned(XMLNode.ChildNodes) then
  for i := 0 to XMLNode.ChildNodes.Count - 1  do
   SetXMLNode(NewNode, XMLNode.ChildNodes[i]);
end;
procedure TfmMain.bbOpenClick(Sender: TObject);
var i: Integer;
begin
 if edXMLFileName.Text <> '' then
 begin
  ClDataSet.Close;
  XMLDoc.Active := False;
  tvXMLDoc.Items.Clear;
  XMLProvider.XMLDataFile := edXMLFileName.Text;
  XMLProvider.TransformRead.TransformationFile := edReadFileName.Text;
  XMLProvider.TransformWrite.TransformationFile := edWriteFileName.Text;
  try
   ClDataSet.Open;
   XMLDoc.XML.Clear;
   XMLDoc.XML.Add(XMLClient.GetDataAsXml(edReadFileName.Text));
   XMLDoc.Active := True;
   SetXMLNode(nil, XMLDoc.DocumentElement);
  except
   on E: Exception do
```

```
   begin
    tvXMLDoc.Items.Clear;
    ClDataSet.Close;
    XMLDoc.Active := False;
    Application.MessageBox(pChar(E.Message), pChar(AppTitle),
MB_ICONERROR or MB_OK);
   end;
  end;
 end;
end;
```

В результате при выборе файла XML приложение отображает данные в виде дерева XML и таблицы набора данных.

Примечание

Приложение XML Viewer может работать и с удаленным сервером. Для этого помимо свойства `ProviderName` необходимо задать свойство `RemoteServer` компонентов `TClientDataSet` и `TXMLTransformClient`.

Резюме

Распределенные приложения могут использовать данные в формате XML наравне с другими источниками данных. Более того, во многих случаях для этого пригодны стандартные или минимально модифицированные компоненты. Основную роль здесь играет компонент `TXMLTransformProvider`.

Распределенные приложения баз данных могут работать с данными XML, конвертированными в набор данных, инкапсулированный компонентом `TClientDataSet`.

Данные в виде документа XML, представленные компонентами `TXMLDocument` и `TXMLTransformClient`, доступны любым клиентским приложениям.

Распределенные Internet-приложения имеют возможность публиковать данные XML на Web-страницах.

Для преобразования данных XML в формат пакета данных Delphi (который и позволяет использовать эти данные без особого труда) разработчики могут использовать утилиту XML Mapper.

ЧАСТЬ VIII

Генератор отчетов Rave Reports 6.0

Глава 31

Отчеты Rave Reports в приложении

Генератор отчетов Rave Reports 6 является основным средством создания отчетов для приложений Win32 в Delphi 2005. Генератор отчетов состоит из набора компонентов, визуальной среды создания отчетов и набора библиотек и пакетов. Компоненты Rave Reports устанавливаются в Палитре инструментов в категории **Rave**. Однако для того, чтобы увидеть эту категорию в Delphi, разработчику нужно самостоятельно инсталлировать пакет компонентов Rave Reports в Палитру инструментов. Для этого достаточно при помощи команды **Component | Install Packages** главного меню открыть диалог добавления новых пакетов и добавить пакет dclRave.bpl из папки ..\Rave Reports\Lib.

Кроме этого, для удобства работы желательно добавить в меню **Tools** главного окна Delphi команду для вызова визуальной среды создания отчетов Rave Reports.

В этой главе рассматриваются следующие вопросы.

- Какие компоненты входят в состав Rave Reports и на какие функциональные группы они делятся.
- Что такое проект отчета и его структура.
- Как включить отчет в состав приложения и какие компоненты для этого необходимы.
- Компоненты управления отчетами.

Генератор отчетов Rave Reports 6

Генератор отчетов Rave Reports 6 разработан фирмой Nevrona и входит в состав Delphi 2005 в качестве основного средства для создания отчетов для приложений Win32. Он состоит из трех частей.

- Ядро генератора отчетов обеспечивает управление отчетом, его предварительный просмотр и отправку на печать. Исполняемый код ядра сервера

включается в приложение Delphi, делая его полностью автономным при работе с отчетами на компьютере клиента.

- Визуальная среда разработки отчетов Rave Reports предназначена для разработки самих отчетов. Она позволяет добавлять к отчету страницы, размещать на них графические и текстовые элементы управления, подключать к отчетам источники данных и т. д. Отчеты сохраняются в файлах с расширением rav и должны распространяться совместно с приложениями, использующими их.
- Компоненты Rave Reports расположены в категории **Rave** Палитры инструментов Delphi. Они обеспечивают управление отчетами в приложении.

Генератор отчетов устанавливается при инсталляции Delphi в папку ..\Rave Reports. Исходные коды компонентов разработчикам в Delphi недоступны.

Безусловно, визуальная среда разработки заметно упрощает процесс создания отчетов и позволяет добиться лучших результатов меньшими усилиями. Тем не менее при первом знакомстве с продуктом заметны и его недостатки. Система помощи оставляет тягостное впечатление не только своей крайней лаконичностью, но и фактическими ошибками. Многие свойства и методы остались недокументированными, и наоборот — имеющиеся в статьях подсказки описания не имеют реальных аналогов в коде компонентов.

Компоненты Rave Reports и их назначение

Компоненты для создания и управления отчетами делятся на следующие функциональные группы.

- Компонент отчета `TRvProject` с точки зрения приложения и есть отчет. Он обеспечивает загрузку заранее созданного в визуальной среде Rave Reports отчета из файла с расширением rav.

  Подробнее об использовании компонента `TRvProject` рассказывается далее в этой главе.

- Компонент управления отчетом `TRvSystem` обеспечивает работу приложения с отчетом. Взаимодействуя с компонентом отчета с одной стороны, и сервером отчета Rave Reports с другой, этот компонент обеспечивает просмотр и печать отчетов.

  Подробнее об использовании компонента `TRvSystem` рассказывается далее в этой главе.

- Компоненты соединения с источниками данных предназначены для подключения различных источников данных к отчетам. При этом могут использоваться технологии доступа к данным dbGo, BDE, dbExpress *(см. часть V)*.

  К этой группе относятся компоненты:

  - `TRvCustomConnection`;
  - `TRvDataSetConnection`;
  - `TRvTableConnection`;
  - `TRvQueryConnection`;

  Подробнее об использовании этих компонентов рассказывается в *главе 34*.

- Компоненты преобразования данных позволяют конвертировать отчеты из формата данных Rave Reports в другие форматы (текстовый, PDF, HTML, RTF), а также распечатывать или просматривать отчеты.

  К этой группе относятся компоненты:

  - `TRvNDRWriter`;
  - `TRvRenderPreview`;
  - `TRvRenderPrinter`;
  - `TRvRenderPDF`;
  - `TRvRenderHTML`;
  - `TRvRenderRTF`;
  - `TRvRenderText`;

  Подробнее об использовании компонентов преобразования данных рассказывается далее в этой главе.

Отчет в приложении Delphi

Завершив обзор нового генератора отчетов, давайте обратимся к деталям программирования и посмотрим, что нужно сделать, чтобы приложение могло работать с отчетами.

Как уже упоминалось, основой отчета является файл отчета с расширением rav. Он создается в визуальной среде разработки Rave Reports и может содержать произвольное число страниц. Каждая страница может быть оформлена графическими или текстовыми элементами или отображать данные из какой-либо базы данных. Другими словами файл RAV — это проект будущего отчета, содержащий общую информацию об отчете, оформление его страниц и правила их заполнения.

После создания проект отчета необходимо связать с приложением Delphi. Для этого используется компонент `TRvProject` (рис. 31.1), который обеспечивает представление отчета в приложении.

Но этого недостаточно, чтобы просмотреть или напечатать отчет. Для выполнения этих операций к исполняемому коду приложения автоматически прикомпилируется код ядра генератора отчета при переносе на любую форму проекта компонентов `TRvProject` и `TRvSystem`. Для управления операциями печати и просмотра в проекте должен присутствовать компонент `TRvSystem` (см. рис. 31.1).

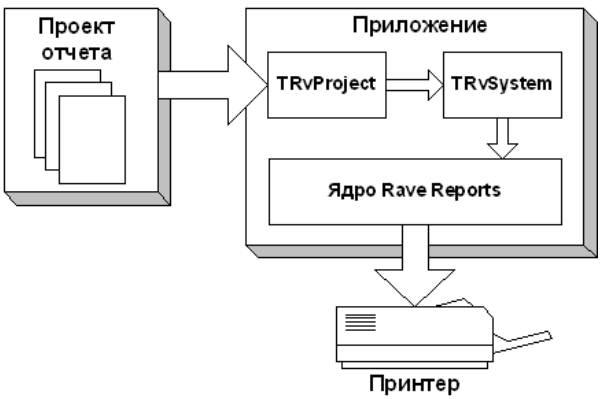


Рис. 31.1. Компоненты Rave Reports в приложении Delphi

Для того чтобы приложение Delphi могло выполнять функции печати отчетов, разработчик должен выполнить следующий набор операций.

1. При помощи визуальной среды разработки Rave Reports необходимо создать проект отчета и сохранить его *(см. главу 32)*.
2. Перенести в проект приложения в Delphi компонент `TRvProject` и связать его с файлом проекта отчета (см. далее в этой главе) при помощи свойства `ProjectFile`.
3. Перенести в проект приложения в Delphi компонент `TRvSystem` и связать его с компонентом `TRvProject`. Для этого имеется свойство `Engine` компонента `TRvProject` (см. далее в этой главе).
4. Написать код приложения, обеспечивающий просмотр и печать отчета (при необходимости и другие операции), используя методы компонента `TRvProject` (см. далее в этой главе).

Конечно же, это наиболее простой способ включения отчета в приложения. Для решения более сложных задач необходимо изучить описанные компоненты более детально.

Компонент отчета *TRvProject*

Компонент `TRvProject` обеспечивает представление в приложении отчета. Для связи проекта отчета Rave Reports с компонентом служит свойство

```
property ProjectFile: string;
```

До начала печати необходимо связать компонент `TRvProject` с компонентом управления отчетом `TRvSystem`. Для этого достаточно передать в свойстве

```
property Engine: TRpComponent;
```

ссылку на компонент `TRvSystem`. При необходимости вы можете загрузить отчет из внешнего файла или потока:

```
procedure LoadFromFile(FileName: String);procedure LoadFromStream(Stream:
TStream);
```

Загруженный отчет становится текущим.

Кроме этого, существует и пара методов для сохранения отчета:

```
function SaveToFile(FileName: String);procedure SaveToStream(Stream:
TStream);
```

В процессе работы приложения может потребоваться напечатать несколько различных отчетов. Для этого можно вызвать потребное число компонентов `TRvProject` или загружать нужные отчеты по мере необходимости.

Кроме этого, забегая немного вперед *(см. главу 32)*, скажем, что один файл проекта отчета может содержать несколько независимых отчетов. Каждый из них идентифицируется в компоненте `TRvProject` тремя свойствами. Имя, полное имя и описание отчета содержатся соответственно в трех свойствах только для чтения:

- `property ReportName: String;`
- `property ReportFullName: String;`
- `property ReportDesc: String;`

Эти три свойства возвращают параметры текущего отчета. Сразу после загрузки из файла текущим становится отчет, являющийся отчетом по умолчанию в среде разработки. При необходимости сменить текущий отчет применяется метод

```
function SelectReport(ReportName: string; FullName: boolean): boolean;
```

В параметре `ReportName` передается имя нужного отчета. Если параметр `FullName` имеет значение `True`, то это полное имя отчета, иначе — имя отчета.

В случае если проект содержит несколько отчетов, их имена доступны при помощи метода

```
procedure GetReportList(ReportList: TStrings;FullName: boolean);
```

Список имен будет возвращен в список строк `ReportList`, а параметр `FullName` определяет, какие именно имена будут занесены в список. При значении параметра `True` метод возвращает полные имена отчетов (соответствует свойству `ReportFullName`), иначе — имена (соответствует свойству `ReportlName`).

Например, код

```
var ReportList: TStringList;
    i: Integer;
...
 ReportList := TStringList.Create;
 RvProject1.Open;
 try
  RvProject1.GetReportList(ReportList, False);
  for i := 0 to ReportList.Count – 1
   do RvProject1.ExecuteReport(ReportList[I]);
 finally
  RvProject1.Close;
  ReportList.Free;
 end;
```

последовательно печатает все отчеты, входящие в состав файла проекта отчета. Файл проекта отчета можно включить в состав исполняемого файла приложения. Для этого используется свойство

```
property StoreRAV: Boolean;
```

При щелчке на кнопке в строке этого свойства в Инспекторе объектов открывается специализированный редактор **Load Into EXE** (рис. 31.2).

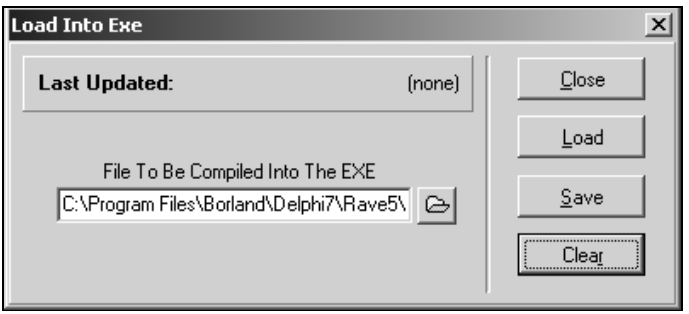


Рис. 31.2. Редактор свойства `StoreRAV` компонента `TRvProject`

Здесь можно задать файл проекта отчета. После этого в Инспекторе объектов в строке свойства StoreRAV появятся дата и время загрузки проекта отчета. Эти же время и дата будут сохранены в свойстве

```
property RaveBlobDateTime: TDateTime;
```

Отправить отчет на печать можно методом

```
procedure Execute;
```

или же методом

```
procedure ExecuteReport(ReportName: string);
```

который позволяет направить на печать отчет, заданный параметром ReportName. Он должен соответствовать имени отчета, хранящегося в свойстве ReportName компонента TRvProject.

Отчет, содержащийся в компоненте TRvProject, может быть открыт для редактирования методом

```
procedure Open;
```

Не открывая отчет, вы не сможете работать с большинством свойств и методов компонента. Дело в том, что при открытии компонент загружает отчет из файла проекта или прикомпилированного кода (в случае свойства StoreRAV).

Сохранение и закрытие отчета соответственно выполняются методами

```
procedure Save;procedure Close;
```

Кроме этого, действия аналогичные методам Open и Close, выполняются свойством

```
property Active: Boolean;
```

Если свойству присвоить значение True — отчет открывается, иначе — закрывается.

До и после открытия и закрытия отчета вызывается четверка методов-обработчиков:

```
property BeforeOpen: TNotifyEvent;property AfterOpen:
TNotifyEvent;property BeforeClose: TNotifyEvent;property AfterClose:
TNotifyEvent;
```

Компонент управления отчетом *TRvSystem*

Компонент управления отчетом TRvSystem обеспечивает выполнение основных операций с отчетом из приложения. В приложении он должен быть связан с компонентом TRvProject. Этого вполне достаточно, чтобы компо-

нент TRvSystem выполнил свою работу. У разработчика нет необходимости вызывать какие-либо методы компонента, чтобы направить отчет на печать.

В его составе инкапсулированы объекты, обеспечивающие вывод отчета из компонента TRvProject в один из трех системных приемников:

- файл (объект класса TSystemFiler);
- предварительный просмотр (объект класса TSystemPreview);
- принтер (объект класса TSystemPrinter).

За это отвечает свойство

```
type
  TReportDest = (rdPreview, rdPrinter, rdFile);property ReportDest:
  TReportDest;
```

которое может принимать одно из трех значений типа TReportDest. Соответственно, для каждого типа системного приемника имеется свойство, позволяющее задать все его основные параметры.

Для вывода в файл это комплексное свойство

```
property SystemFiler: TSystemFiler;
```

Внутри него задается имя файла во вложенном свойстве

```
property FileName: string;
```

но при этом вложенное свойство

```
type
  TStreamMode = (smMemory, smTempFile, smFile, smUser);property
  StreamMode: TStreamMode;
```

должно иметь значение smFile. Для вывода отчета для предварительного просмотра предусмотрен экземпляр класса TSystemPreview, который доступен через свойство

```
property SystemPreview: TSystemPreview;
```

Его свойства совпадают со свойствами компонента TRvRenderPreview. Стандартное диалоговое окно предварительного просмотра отчета Rave Report представлено на рис. 31.3.

Заголовок этого окна задается свойством

```
property TitlePreview: TFormatString;
```

Перед открытием окна предварительного просмотра вызывается метод-обработчик

```
property OnPreviewShow: TNotifyEvent;
```

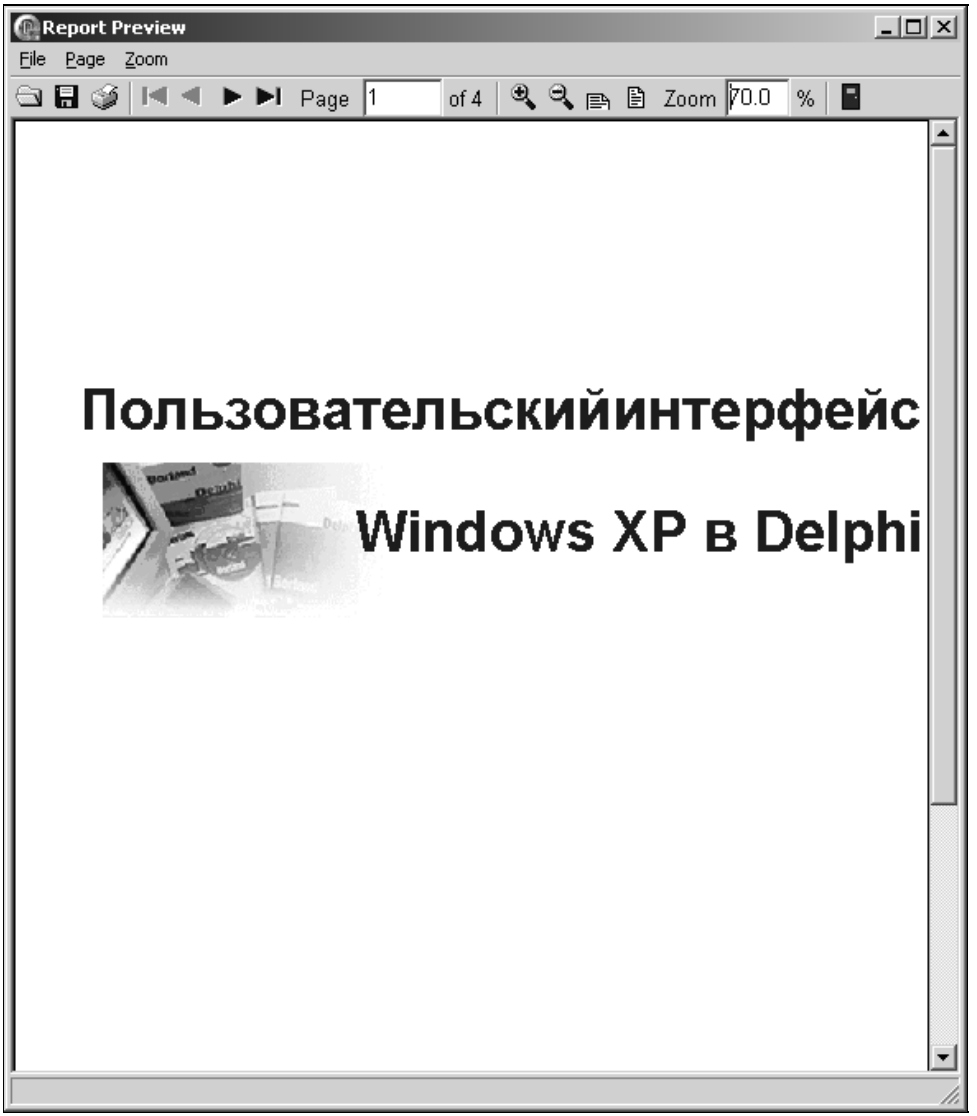


Рис. 31.3. Стандартное диалоговое окно предварительного просмотра компонента `TRvSystem`

За вывод отчета на печать отвечает инкапсулированный в компоненте объект типа `TSystemPrinter`. К нему можно обратиться при помощи свойства

```
property SystemPrinter: TSystemPrinter;
```

Его свойства совпадают со свойствами компонента `TRvRenderPrinter`.

Перед тем, как отправить отчет одному из трех системных приемников, компонент открывает диалог настройки печати (рис. 31.4).

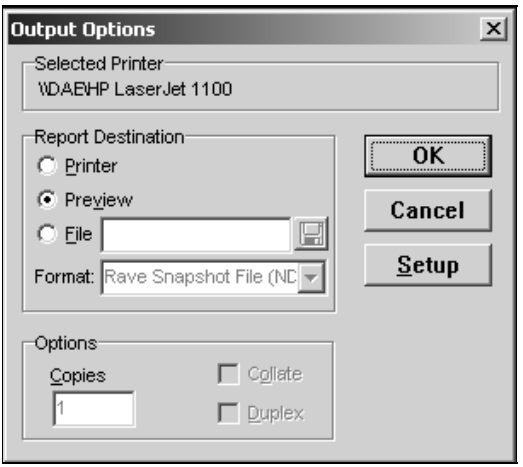


Рис. 31.4. Диалог настройки печати компонента `TRvSystem`

Его заголовок определяется свойством

```
property TitleSetup: TFormatString;
```

Перед открытием этого окна вызывается метод-обработчик

```
property OnPreviewSetup: TNotifyEvent;
```

Кроме этого, для диалога настройки печати можно задать ряд дополнительных параметров. Это делается в свойстве

```
type
  TSystemSetup = (ssAllowSetup, ssAllowCopies, ssAllowCollate,
  ssAllowDuplex, ssAllowDestPreview, ssAllowDestPrinter, ssAllowDestFile,
  ssAllowPrinterSetup);  TSystemSetups = set of TSystemSetup;property
  SystemSetups: TSystemSetups;
```

Назначение элементов множества `TSystemSetup` следующее:

- `ssAllowSetup` — разрешает или запрещает диалог настройки печати компонента;
- `ssAllowCopies` — управляет доступностью установки числа копий отчета;
- `ssAllowCollate` — разрешает или запрещает настройку режима печати с разбором страниц по копиям;
- `ssAllowDuplex` — разрешает или запрещает настройку двусторонней печати;

- `ssAllowDestPreview` — разрешает или запрещает вызов окна предварительного просмотра;
- `ssAllowDestPrinter` — разрешает или запрещает использование принтера;
- `ssAllowDestFile` — разрешает или запрещает использование файла для вывода отчета;
- `ssAllowPrinterSetup` — разрешает или запрещает диалог настройки параметров принтера.

Во время выполнения любой из перечисленных операций вывода отчета открывается окно состояния процесса (рис. 31.5). Его заголовок определяется свойством

```
property TitleStatus: TFormatString;
```

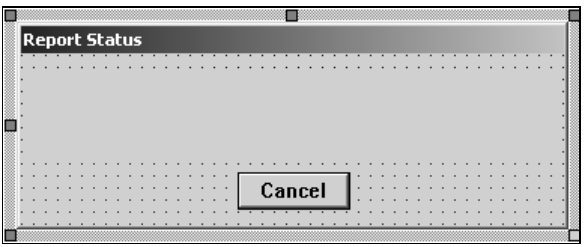


Рис. 31.5. Форма состояния процесса вывода отчета компонента `TRvSystem`

В нем отображается информационная строка состояния, которая может быть настроена при помощи свойств объекта `SystemFiler`, представленного в компоненте `TRvSystem` одноименным свойством.

Вложенное свойство

```
property StatusFormat: string;
```

определяет строку форматирования для текста о состоянии процесса.

Для нее предусмотрены следующие управляющие символы:

- `%c` — текущее состояние процесса вывода;
- `%p` — номер текущей страницы;
- `%f` — номер первой страницы;
- `%l` — номер последней страницы;
- `%d` — название устройства вывода (название принтера, имя файла, предварительный просмотр);
- `%r` — имя драйвера устройства вывода;
- `%s` — общее число страниц;

- `%t` — порт печати;
- `%0—%9` — номера строк для свойства `StatusText`.

Вложенное свойство

```
property StatusText: TStrings;
```

позволяет задать до десяти строк (можно задать и больше, но они не будут восприняты строкой статуса) с какой-либо дополнительной информацией, описывающей процесс вывода. Первая строка списка будет выведена при наличии в свойстве `StatusFormat` управляющего символа `%0`, вторая — при наличии символа `%1` и т. д. При помощи перечисленных свойств вы сможете детально описать процесс вывода отчета. В этом вам помогут методы-обработчики событий компонента `TRvSystem`. До начала печати отчета и после его окончания (даже если печать была прервана) соответственно вызывается пара методов-обработчиков:

```
property OnBeforePrint: TNotifyEvent;
property OnAfterPrint: TNotifyEvent;
```

При начале печати непосредственно отчета (не заголовка) вызывается метод-обработчик

```
property OnPrint: TNotifyEvent;
```

Если вы печатаете одну страницу, будет вызван метод-обработчик

```
type
  TPrintPageEvent = function(Sender: TObject; var PageNum: Integer):
  Boolean;
property OnPrintPage: TPrintPageEvent;
```

Но до начала печати вызывается метод-обработчик

```
property OnNewPage: TNotifyEvent;
```

который обозначает генерацию страницы.

При печати колонтитулов в верхней и нижней частях страницы вызываются методы-обработчики

```
property OnPrintHeader: TNotifyEvent;
property OnPrintFooter: TNotifyEvent;
```

Разработчик может задать несколько опций для всего компонента `TRvSystem`, управляя тем самым процессом вывода отчета. Для этого используется свойство

```
type
  TSystemOption = (soUseFiler, soWaitForOK, soShowStatus,
  soAllowPrintFromPreview, soPreviewModal);  TSystemOptions = set of
  TSystemOption;property SystemOptions: TSystemOptions;
```

А элементы типа `TSystemOptions` обозначают следующее:

- `soUseFiler` — при установке этой опции в `True` вывод будет направляться в файл, заданный свойством `SystemFiler`, независимо от других настроек компонента;
- `SoWaitForOK` — если включить эту опцию, генерация отчета будет задержана до момента, когда пользователь нажмет кнопку **OK** в диалоге настройки печати компонента (см. рис. 31.4);
- `soShowStatus` — эта опция управляет видимостью окна состояния процесса вывода отчета в компоненте;
- `soAllowPrintFromPreview` — будучи включенной, эта опция позволяет печатать отчет из окна предварительного просмотра;
- `soPreviewModal` — при значении `True` делает окно предварительного просмотра модальным.

Резюме

В качестве основного средства создания отчетов и их использования в приложениях в состав Delphi 2005 включен генератор отчетов Rave Reports 6. В его состав входят ядро генератора отчетов, визуальная среда создания отчетов и набор компонентов.

Ядро генератора отчетов обеспечивает предварительный просмотр или печать отчета. Оно включается в исполняемый файл приложения. Поэтому разработчики избавлены от необходимости распространять совместно с приложением какие-либо дополнительные файлы.

Визуальная среда создания отчетов позволяет разрабатывать самые разнообразные отчеты, в том числе работающие с наборами данных из источников различных типов.

Набор компонентов предоставляет разработчику инструментарий для управления отчетом в приложении.

Глава 32

Визуальная среда создания отчетов

Визуальная среда создания отчетов входит в состав генератора отчетов Rave Reports 6. Она значительно облегчает самый трудоемкий этап в процессе создания отчета и его включения в состав приложения — постраничную разработку шаблона отчета.

Под *шаблоном отчета* мы подразумеваем совокупность страниц отчета с расположенными на них графическими и текстовыми элементами оформления, а также свойствами и правилами создания отчета, сохраненными в файле с расширением rav.

Каждый файл RAV может включать несколько независимых шаблонов отчетов. Затем эти шаблоны используются в компонентах `TRvProject` и служат основой для создания и печати отчетов в приложении.

Кроме этого, средствами визуальной среды шаблон отчета может быть подключен к различным источникам данных. При этом возможны следующие технологии доступа к данным:

- ADO
- dbExpress
- BDE
- InterBase

Визуальная среда Rave Reports открывается командой **Rave Visual Designer** всплывающего меню компонента `TRvProject`. Кроме этого, можно добавить новую команду в меню **Tools** главного окна Delphi. Для этого необходимо использовать команду **Tools | Configure Tools**.

Выделим несколько основных задач, которые можно решать в визуальной среде создания отчетов.

- Загрузка, редактирование и сохранение шаблонов отчетов в файлах RAV.
- Создание структуры отчета и определение его основных свойств.
- Разработка страниц отчета.

- Подключение к отчету источников данных и использование их данных при оформлении страниц отчетов.
- Генерация отчета на основе созданного шаблона, его предварительный просмотр или печать.

В этой главе рассматриваются следующие вопросы.

- Составные части визуальной среды и инструментарий создания отчетов.
- Структура шаблона отчета.
- Текстовые и графические элементы оформления страниц отчетов.
- Использование методов-обработчиков событий.
- Виды источников данных в отчетах и работа с ними при оформлении страниц.

Инструментарий визуальной среды создания отчетов

Пользовательский интерфейс визуальной среды создания отчетов Rave Reports во многом напоминает среду разработки Delphi (рис. 32.1). В верхней части окна располагается панель инструментов, состоящая из набора кнопок (слева) и Палитры инструментов (справа). В Палитре инструментов располагаются не только элементы оформления отчетов, но и инструменты для их настройки и управления.

Давайте посмотрим, для чего предназначены закладки Палитры инструментов. Первые четыре содержат элементы оформления отчетов:

- **Drawing** — графические элементы оформления;
- **Bar Code** — различные типы штрихкодов;
- **Standard** — элементы оформления, позволяющие размещать на страницах отчета текст и изображения;
- **Report** — элементы оформления, предназначенные для отображения данных из внешних источников данных, подключенных к отчету.

Остальные закладки содержат инструменты управления и настройки страниц и элементов оформления:

- **Zoom** — управляет увеличением текущей страницы;
- **Colors** — позволяет установить цвета элементов оформления и страниц;
- **Lines** — задает стиль и толщину линий элементов оформления;
- **Fills** — задает стиль заполнения элементов оформления;

- **Fonts** — позволяет задать параметры шрифта для текста;
- **Alignment** — управляет выравниванием элементов оформления на странице.

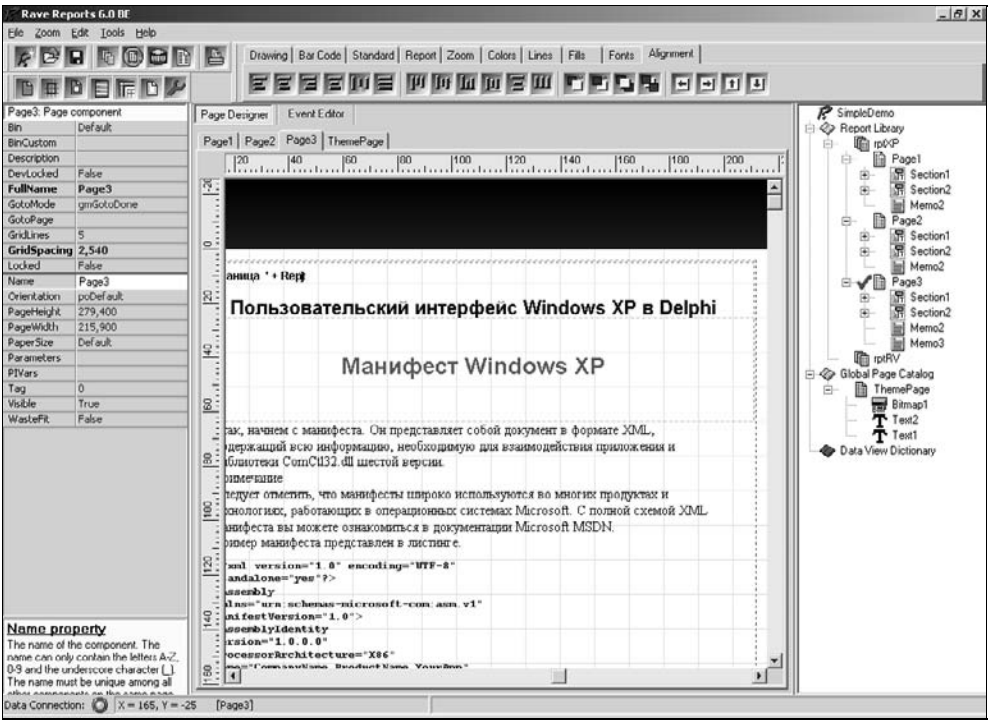


Рис. 32.1. Визуальная среда создания отчетов Rave Reports

Центральную часть окна занимает блокнот с двумя закладками.

Закладка **Page Designer** содержит еще один блокнот, каждая из страниц которого соответствует одной странице отчета. Когда вы добавляете к отчету новую страницу, здесь появляется еще одна закладка с именем новой страницы. На страницы можно переносить элементы оформления, изменять их размеры и местоположение. На страницу также можно спроектировать измерительную сетку, которая поможет размещать и выравнивать элементы оформления. Обрамляют страницу вертикальная и горизонтальная линейки.

На страницу можно переносить элементы оформления из Палитры инструментов, и затем выделять их, настраивать их свойства, перемещать и удалять.

Закладка **Event Editor** обеспечивает создание методов-обработчиков событий для отчетов, страниц, элементов оформления и т. д.

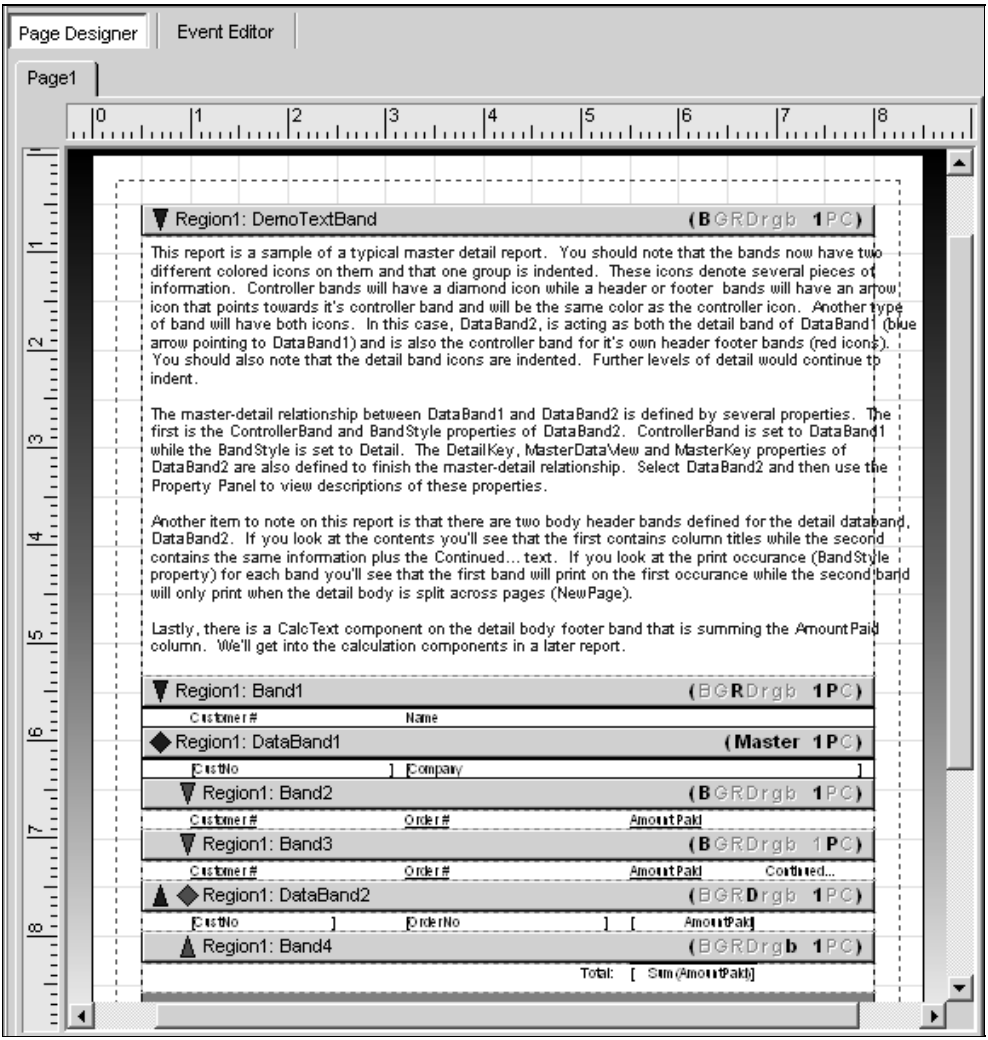


Рис. 32.2. Закладка **Page Designer** визуальной среды создания отчетов Rave Reports

Правую часть окна среды разработки занимает панель проекта отчета. Дерево проекта содержит все его составные части. При двойном щелчке на элементе дерева он отображается на странице в центральной части.

В левой части окна среды разработки располагается аналог Инспектора объектов Delphi, в котором доступны свойства текущего элемента. В нижней части этой панели отображается подсказка для текущего свойства.

Проект отчета

Визуальная среда работает с проектом отчета, который создается или загружается из файла с расширением rav. Состав проекта отчета отображается в дереве проекта отчета в панели в правой части окна визуальной среды (рис. 32.3).

Корневой элемент Rave Project содержит три дочерние ветви.

- **Report Library** — библиотека отчетов включает все шаблоны отчетов, содержащиеся в этом проекте.
- **Global Page Catalog** — каталог глобальных страниц содержит перечень страниц, не принадлежащих какому-либо из отчетов проекта.
- **Data View Dictionary** — словарь просмотров данных содержит созданные соединения с внешними источниками данных.

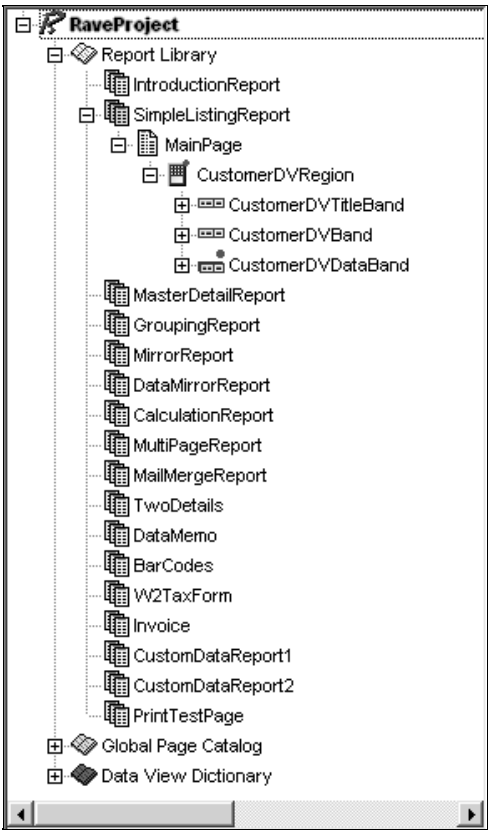


Рис. 32.3. Дерево проекта отчета

А теперь давайте посмотрим, из каких составных частей может состоять проект отчета.

Библиотека отчетов

В первую очередь это отчеты, входящие в состав библиотеки отчетов. Каждый из этих отчетов описывает отдельный, самостоятельный отчет. Любой из них может быть загружен в компонент `TRvProject` для применения в приложениях Delphi. Первый отчет в списке по умолчанию становится текущим. Для смены текущего отчета достаточно дважды щелкнуть на нем в дереве проекта и это состояние будет сохранено при закрытии проекта.

Добавить к проекту новый отчет можно кнопкой в главной панели окна визуальной среды или командой **File | New Report** главного меню. Для удаления отчета достаточно сделать его текущим и нажать клавишу <Delete>.

Для каждого отчета необходимо заполнить свойства `Name` и `FullName`, которые служат для идентификации отчета при работе с ним в Delphi *(см. главу 31)*. Кроме этого, в свойстве `Description` полезно заполнить описание отчета и задать единицы измерения, т. к. по умолчанию установлены дюймы.

Рис. 32.4. Редактор страниц отчета **Page List Editor**

Отчет может содержать произвольное число страниц. Напечатать можно как все страницы, так и их произвольное подмножество. Для разработчика спи-

сок страниц отчета, которые предлагаются к печати по умолчанию, доступен в свойстве `PageList`. С этим свойством связан редактор страниц **Page List Editor** (рис. 32.4), который позволяет выбирать страницы отчета и формировать из них список для печати. При этом одна страница может быть включена в список несколько раз.

Для добавления к текущему отчету новой страницы используйте кнопку в главной панели окна визуальной среды или команду **File | New Report Page** главного меню. Для удаления выберите страницу и нажмите клавишу <Delete>.

Страница имеет имя, задаваемое свойством `Name`, а также несколько свойств, задающих ее важнейшие параметры: `Orientation`, `PageSize`, `PageHeight` и `PageWidth`.

Свойство `GotoPage` позволяет задать страницу, которая будет напечатана после этой. А порядок печати страниц по умолчанию соответствует их порядку в дереве отчета.

Свойство `GridLines` позволяет задать плотность измерительной сетки, накладываемой на страницу в визуальной среде для удобства размещения элементов оформления.

Каталог глобальных страниц

Каталог глобальных страниц объединяет страницы, доступные из любого отчета библиотеки отчетов. Таким образом, вы можете оформить все отчеты проекта одинаково. Например, для всех отчетов можно создать глобальные страницы титульного листа, общего заголовка и т. д.

Добавить новую страницу можно при помощи команды главного меню **File | New Global Page**. После этого страница появляется в списке каталога и доступна для редактирования.

Глобальная страница добавляется в отчет при помощи редактора страниц (см. рис. 32.4). Для этого необходимо выбрать нужную страницу из списка **Global Pages** и нажать на кнопку **Add Global**.

Словарь просмотра данных

Словарь просмотра данных объединяет разнообразные объекты доступа к данным. Для создания нового объекта необходимо воспользоваться командой главного меню **File | New Data Object**. После этого открывается диалог выбора типа объекта **Data Connections** (рис. 32.5).

Здесь доступны следующие типы объектов.

- **Data Lookup Security Controller** — организует аутентификацию пользователя по имени и паролю при использовании одного из просмотра данных.

- **Database Connection** — создает соединение с внешним источником данных на основе одной из трех технологий доступа к данным: ADO, BDE и dbExpress.
- **Direct Data View** — создает просмотр данных на основе активного соединения с источником данных.
- **Driver Data View** — создает просмотр данных на основе ранее созданного в словаре соединения.
- **Simple Security Controller** — представляет собой список пользователей, который может быть использован для организации доступа в отдельных отчетах.

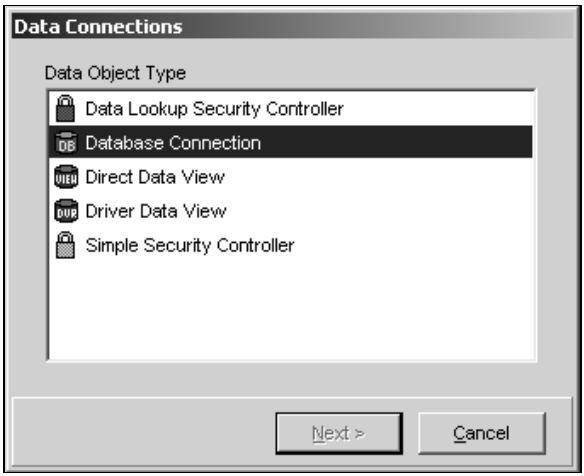


Рис. 32.5. Диалог выбора типа объекта доступа к данным **Data Connections**

Созданные в словаре объекты являются глобальными для всего проекта и доступны в любой странице любого отчета. Мы познакомимся с ними более подробно дальше в этой главе.

Стандартные элементы оформления и их свойства

Теперь остановимся подробнее на элементах оформления отчетов. Они используются так же, как и компоненты в Delphi. Выбранный элемент переносится из Палитры инструментов на страницу отчета. Здесь его можно разместить в нужном месте, изменить его размеры и настроить свойства.

Свойства элемента оформления доступны в панели слева. Кроме этого, наиболее важные визуальные свойства (цвет, стиль линий и заполнения и т. д.) вынесены в Палитру инструментов.

После переноса на страницу имя элемента оформления также появляется в дереве проекта.

Элементы для представления текста и изображений

В категории **Standard** Палитры инструментов расположены элементы оформления, предназначенные для отображения текста и изображений. Рассмотрим их.

Для представления однострочного текста имеется простой элемент оформления `Text`. Собственно текст задается свойством `Text`. Другие стандартные свойства позволяют настраивать шрифт, цвет и т. д. Кроме этого, свойство `Rotation` позволяет повернуть текст на любой угол в диапазоне от 0 до 360 градусов.

Для представления многострочного текста предусмотрен элемент оформления `Memo`. Текст задается свойством `Memo`.

Если вам необходимо объединить несколько элементов оформления в группу (например, несколько строк текста и изображений в заголовке страницы) и разместить их на странице совместно, удобен невизуальный элемент оформления `Section`. Размещенные на нем другие элементы как бы оказываются на самостоятельной страничке, ограниченной элементом `Section`. Вы можете выделять и перемещать отдельные элементы, но делать это только внутри секции. А при перемещении секции по странице все расположенные на ней элементы также перемещаются вместе с ней. Выравнивание элементов также работает по границам секции.

Для представления изображений есть элемент оформления `Bitmap`. Он позволяет оформлять отчеты изображениями, сохраненными в файлах в формате BMP. Для загрузки изображения используется диалог, открывающийся при щелчке на кнопке свойства `Image`. Также при помощи свойства `FileLink` можно связать элемент с файлом изображения и при печати отчета оно будет загружено. Кроме этого, изображение можно загрузить из базы данных. Для этого существуют свойства `DataView` (определяет объект просмотра данных) и `DataField` (задает поле изображения).

Изображение можно масштабировать. Для этого имеется свойство `MatchSide`. При его значениях `msHeight`, `msWidth`, `msBoth` изображение соответственно масштабируется по горизонтали, вертикали или в двух измере-

ниях. Естественно при этом может произойти искажение изображения. А вот при значении `msInside` изображение будет отмасштабировано пропорционально.

Аналогичными свойствами обладает элемент оформления `MetaFile`. Но изображение в формате WMF загружается в него при помощи свойств `FileLink` или `DataField`.

Невизуальный элемент оформления `FontMaster` позволяет задать единые свойства шрифта для группы элементов оформления. Его свойство `Font` определяет шрифт. И этот шрифт будет установлен для всех элементов, в чьих свойствах `FontMirror` будет указан данный элемент `FontMaster`.

Невизуальный элемент оформления `PageNumInit` обеспечивает нумерацию страниц, начиная с той, на которой он расположен. Свойство `InitValue` задает номер, с которого начинается отсчет нумерации. А при помощи свойств `InitDataView` и `InitDataField` можно загрузить это значение из базы данных.

Графические элементы

Графические элементы для оформления отчетов расположены на странице **Drawing** Палитры инструментов. Конечно, с их помощью вам не удастся изобразить картину Сальвадора Дали, но для оформления отчетов и рисования таблиц они вполне подойдут.

Прежде всего, это три элемента оформления для рисования линий:

- `HLine` — горизонтальная линия;
- `VLine` — вертикальная линия;
- `Line` — универсальная линия.

Все эти элементы имеют идентичный набор свойств, позволяющих задавать параметры и размер линий. Кроме этого, элемент `Line` позволяет развернуть линию на любой угол. Это можно сделать при помощи мыши.

Элементы оформления `Square` и `Rectangle` изображают квадрат и прямоугольник соответственно.

Элементы `Ellipse` и `Circle` изображают эллипс и круг.

Штрихкоды

На странице **Bar Code** разработчику доступны шесть элементов оформления, позволяющие включать в отчеты штрихкоды. Все они реализуют различные стандарты, но значение для кодирования у всех задается одним свойством `Text`. Только элементы `PostNetBarCode`, `I2of5BarCode`, `UPCBarCode` и

`EANBarCode` позволяют вводить только числа, а элементы `Code39BarCode` и `Code128BarCode` могут работать и с буквенно-цифровыми последовательностями.

- Код PostNet используется почтовой службой США, содержит код адреса.
- Перемежающийся код I2of5 служит для представления числовых последовательностей. Перемежающимся назван потому, что цифры в последовательности попеременно кодируются штрихами и пробелами.
- Код Code39 предназначен для кодирования цифр, заглавных букв латинского алфавита и некоторых других символов. Для представления символа используются пять штрихов и четыре пробела.
- Код Code128 позволяет хранить первые 128 символов ASCII.
- Код UPC (Universal Product Code) может содержать только цифры. Разработан для маркировки продуктов. Код может включать 12 цифр.
- Код EAN (European Article Numbering System) подобен UPC. Код может включать 13 цифр. Первые две отводятся под код страны-производителя.

Для всех элементов значение для кодирования можно загрузить из базы данных при помощи свойств `FileLink` или `DataField`.

При необходимости можно рассчитать и напечатать контрольную сумму. Свойство `UseCheckSum` при значении `True` рассчитывает ее, а свойство `PrintCheckSum`, будучи установленным в значение `True`, печатает.

Штрихкод можно развернуть, но только с дискретностью 90 градусов. Для этого есть свойство `BarCodeRotation`.

Обработка событий

Каждому отчету, странице или элементу оформления можно назначить один или несколько методов-обработчиков событий. Для этого предназначен Редактор событий **Event Editor**, доступный через одноименную закладку в центральной части окна визуальной среды Rave Reports.

Для текущего элемента здесь можно выбрать из списка **Available Events** одно из доступных для обработки событий. При этом будет создан обработчик события, программный код которого вводится в окне редактора внизу. В списке **Defined Events** отображаются события, обрабатываемые текущим элементом.

Синтаксис кода для обработки событий аналогичен синтаксису Object Pascal. При этом допустимы некоторые процедуры и функции Delphi.

Например, в метод-обработчик события `BeforePrint` элемента оформления `Text1` можно поместить следующий код:

```
Text1.Text := IntToStr(StrToInt(Text1.Text) + 1);
```

который обеспечит нумерацию страниц отчета.

Проверка созданного кода выполняется при нажатии кнопки **Compile**.

Внешние источники данных в отчете

Все объекты, обеспечивающие доступ к внешним источникам данных из отчетов проекта, собраны в словаре просмотра данных **Data View Dictionary**. Новый объект создается командой **File | New Data Object** главного меню.

Соединение с источником данных и просмотры

В первую очередь необходимо создать соединение с источником данных. Для этого в диалоге (см. рис. 32.5) необходимо выбрать объект соединения **Database Connection** и после нажатия кнопки **Next** в следующем окне выбрать одну из возможных технологий доступа к данным: ADO, dbExpress, BDE. В зависимости от сделанного выбора настраиваются параметры соединения. Подробнее о технологиях доступа к данным рассказывается в главах *части V*.

После завершения настройки готовое соединение появляется в списке **Data View Dictionary**.

На втором этапе к работающему соединению можно подключать объекты просмотров данных. Для этого в диалоге **Data Connections** (см. рис. 32.5) необходимо выбрать **Driver Data View** и нажать кнопку **Next**. Затем в следующем окне нужно выбрать одно из существующих соединений. После этого появляется диалог **Query Advanced Designer** (рис. 32.6).

В этом диалоге необходимо выбрать из списка доступных таблиц источника данных необходимые и задать требуемые соотношения между полями. Страница **Layout** двухстраничного диалога позволяет выбирать таблицы. Нужная таблица выбирается из списка **Tables** в правой части и перетаскивается в левую часть, в которой каждая таблица представляется в виде списка полей. Необходимые поля также можно выбирать для будущего использования. Перетаскиванием полей между таблицами можно задавать внешние ключи.

Страница **Sorting & Grouping** позволяет выбрать поля для сортировки и группировки. Здесь в списке слева представлены выбранные ранее таблицы и поля. Поля можно переносить в список **Sort Fields** справа. Поля в этом

списке будут использованы для сортировки наборов данных. А из полей для сортировки можно выбрать поле для группировки. Для этого используется список **Group By**.

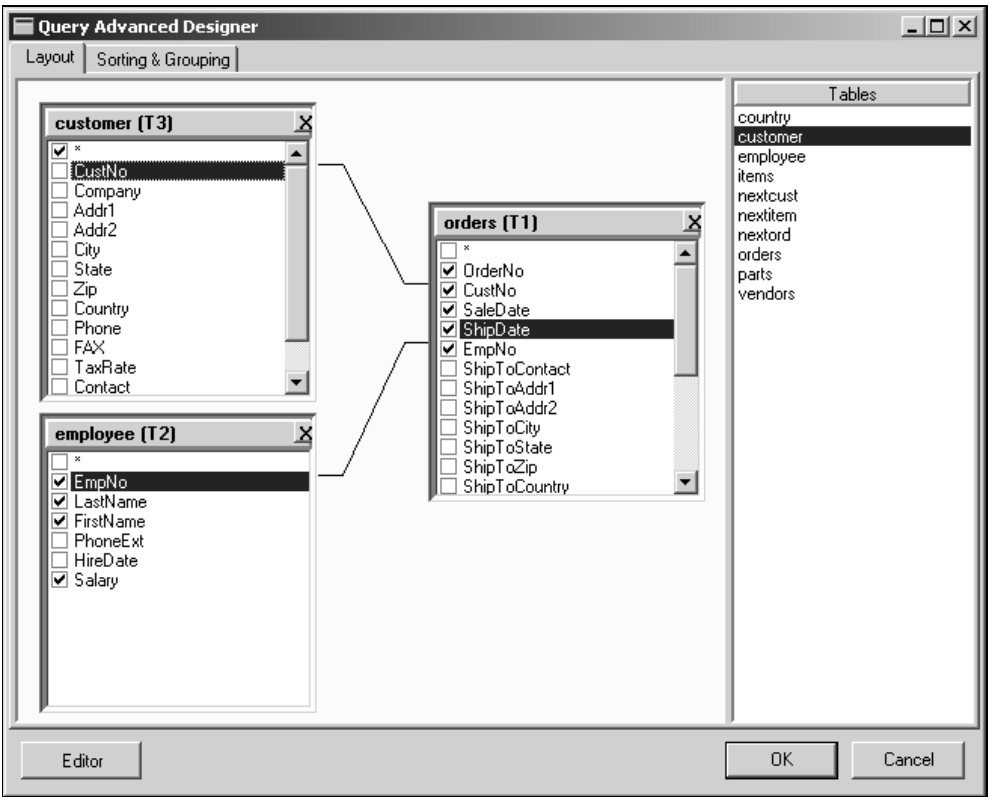


Рис. 32.6. Диалог **Query Advanced Designer**, позволяющий настроить объект просмотра данных

При нажатии на кнопку **Editor** появляется редактор с текстом запроса SQL, реализующего все сделанные ранее настройки. При необходимости его можно исправить вручную.

После завершения настройки просмотра новый объект просмотра появляется в словаре просмотра данных **Data View Dictionary** и может использоваться в отчетах.

Созданный запрос просмотра доступен через его свойство `Query`.

Безопасность доступа к данным

На третьем этапе созданное соединение и просмотр можно защитить. Для этого из списка в диалоге **Data Connections** (см. рис. 32.5) выбирается объект аутентификации **Data Lookup Security Controller**. Он сразу же появляется в словаре просмотра данных **Data View Dictionary**. Остается только подключить его к нужному просмотру. Это обеспечивает его свойство `DataView`, в котором необходимо задать нужный объект просмотра. Списки имен пользователей и их пароли задаются свойствами `UserField` и `PasswordField` соответственно.

Объект аутентификации можно подключить и для использования в отдельном отчете. Для этого используется свойство `SecurityControl` отчета, в котором указывается нужный объект.

Еще один объект — простой объект аутентификации (объект **Simple Security Controller** в диалоге **Data Connections**) в свойстве `UserList` содержит список имен и паролей в формате `UserName=Password`. Он допустим только в отчетах.

Отображение данных в отчетах

Для представления данных в отчетах предназначены специализированные элементы оформления, представленные на странице **Report** Палитры инструментов.

Они делятся на две функциональные группы.

В первую группу выделены элементы, обеспечивающие размножение строк в отчете, их заполнение данными, а также группировку при создании сложных отчетов. Эти элементы мы будем называть *структурными*.

Во второй группе объединены элементы оформления, созданные на основе стандартных, и обеспечивающие отображение текущего значения конкретного поля таблицы просмотра.

Структурные элементы отчета

Рассмотрим структурные элементы.

Основой отчета, использующего просмотры баз данных, является элемент `Region`. Он создает в отчете область, предназначенную для размещения любых других элементов и определяющую часть страницы отчета, отведенную под отображение данных. Он обладает одним интересным свойством `Columns`, которое задает число колонок, в которых будет печататься отчет.

При создании отчета базы данных этот элемент переносится на страницу в первую очередь. Затем приходит очередь элементов `Band` и `DataBand`.

Элемент `Band` создает полосу, на которой можно располагать стандартные элементы оформления. Он служит для оформления заголовков, сносок, врезок и других статичных фрагментов оформления отчетов, которые не изменяются при печати просмотра данных.

Элемент `DataBand` создает полосу, моделирующую строку просмотра данных. На ней располагаются элементы отображения данных, которые будут рассмотрены ниже. При печати отчета для каждой строки печатается новый экземпляр полосы элемента `DataBand` со всеми расположенными на ней элементами оформления. Таким образом и получается отчет, отображающий строка за строкой весь просмотр данных.

Важнейшее свойство `BandStyle` определяет роль и поведение полосы в отчете. С ним связан диалог **Band Style Editor** (рис. 32.7), который отображает взаимосвязь полос в области `Region` отчета и позволяет задать поведение текущей полосы.

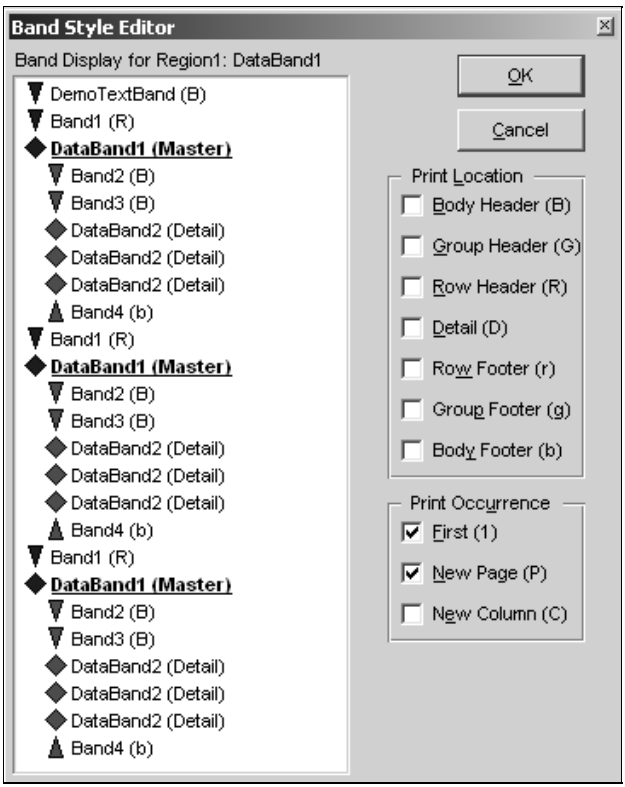


Рис. 32.7. Редактор полос отчета **Band Style Editor**

В левой части диалога отображается список всех полос отчета с их взаимосвязями (отношениями "один ко многим", группировкой, вложенностью и т. д.), текущая полоса выделяется полужирным шрифтом с подчеркиванием. Имя каждой полосы отображается трижды. И это не ошибка разработчиков, а желание показать, что каждая полоса размножается для печати записей просмотра данных.

Группа флажков **Print Location** в правой части диалога определяет назначение полосы:

- **Body Header (B)** — заголовок отчета, печатается в начале отчета;
- **Group Header (G)** — заголовок группы, печатается в начале группы записей, объединенных в просмотре данных выражением GROUP BY;
- **Row Header (R)** — заголовок записи, печатается в начале каждой записи просмотра данных;
- **Detail (D)** — печатается в начале подчиненного набора записей, входящего в отношение "один ко многим";
- **Row Footer (r)** — окончание строки, печатается в конце каждой записи просмотра данных;

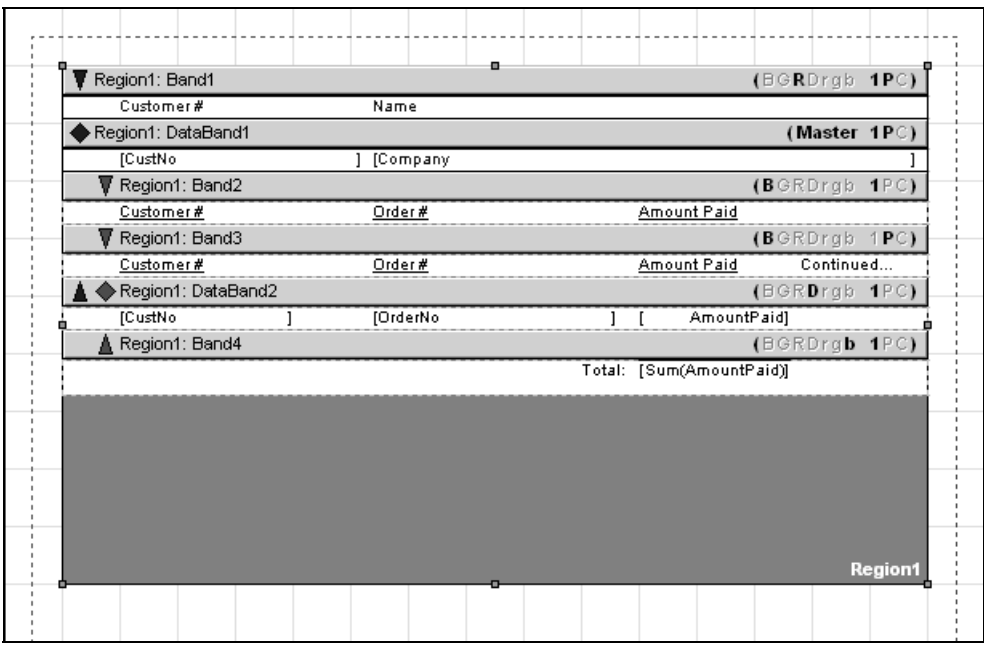


Рис. 32.8. Элементы `Band`, `DataBand` и `Region` на странице отчета

- **Group Footer (g)** — окончание группы, печатается в конце группы записей, объединенных в просмотре данных выражением GROUP BY;
- **Body Footer (r)** — окончание отчета, печатается в его конце.

А группа **Print Occurrence** задает, в каком месте отчета появляется полоса:

- **First (1)** — печатается один раз в начале отчета (титул отчета);
- **New Page (P)** — печатается в начале каждой страницы отчета;
- **New Column (C)** — печатается в начале каждой колонки отчета.

> **Примечание**
>
> Для каждого из перечисленных типов в скобках указан символ, который употребляется для обозначения типа полосы на странице отчета в визуальной среде Rave Reports (рис. 32.8). Таким образом, по совокупности символов разработчик может оценить роль той или иной полосы в отчете, не обращаясь к редактору.

Другие свойства полос и способы создания простых и сложных отчетов рассматриваются в *главе 34*.

Элементы отображения данных

Элементы отображения данных представляют собой модифицированные стандартные элементы, которые размещаются на структурных элементах отчета и отображают данные из связанных с ними полей просмотров данных. Они расположены на странице **Report** Палитры инструментов.

Элемент `DataText` предназначен для представления строковых или числовых значений полей связанного просмотра данных.

Элемент `DataMemo` используется при необходимости показать данные в формате Memo или BLOB.

Элемент `CalcText` обеспечивает выполнение одной из агрегатных функций над значениями связанного поля и представление результата. Тип операции выбирается в свойстве `CalcType`.

Невизуальный элемент `DataMirrorSection`, так же, как и его предок `Section`, объединяет группу других элементов для совместного использования.

Кроме перечисленных, еще один элемент способен отображать данные из поля просмотра. Это стандартный элемент оформления `Bitmap` со страницы **Standard** Палитры инструментов.

Все перечисленные элементы (в том числе и элемент `Bitmap`) связываются с просмотром данных и полем одинаково.

Свойство `DataView` определяет, какой просмотр данных реализуется элементом.

Свойство `DataField` задает поле просмотра, значения которого будут отображаться элементом.

Детально работа с компонентами отображения данных рассматривается в *главе 34*.

Резюме

Визуальная среда Rave Reports позволяет создавать проекты отчетов. Каждый такой проект может содержать несколько отчетов.

Инструментарий визуальной среды позволяет конструировать страницы отчетов из элементов оформления, настраивать их свойства и создавать обработчики событий.

При помощи набора объектов доступа к данным к отчету можно подключить внешний источник данных. При этом возможен выбор одной из трех технологий доступа к данным: ADO, dbExpress и BDE. Специализированный редактор предназначен для визуального построения просмотра данных, а набор элементов отображения данных, используя поля созданного просмотра, обеспечивает создание отчетов.

Глава 33

Разработка, просмотр и печать отчетов

Мы уже обсуждали в предыдущих главах визуальную среду создания отчетов Rave Reports и набор компонентов Delphi в категории **Rave** Палитры инструментов, предназначенный для интегрирования отчета в приложение Delphi. В этой главе мы обратимся к практическим аспектам работы с отчетами Rave Reports в приложениях Delphi.

Рассматриваемые в этой главе примеры не отличаются изощренностью оформления и сложностью, но позволяют поэтапно проследить всю методику создания отчетов и использования их в приложениях и обладают основными атрибутами стандартных отчетов. В них имеются заголовки, нумерация страниц, выводится текстовая и графическая информация.

Но сам по себе отчет — это всего лишь шаблон, который необходимо включить в приложение. Для этого применяются компоненты Rave Reports, которые помимо этой важной функции реализуют еще несколько очень полезных операций. Это, например, возможность сохранения и загрузки отчета в файлах и преобразование типов данных отчета из базового формата RAV в наиболее популярные форматы.

Вопросы создания отчетов для баз данных здесь не затрагиваются и обсуждаются в *главе 34*.

В этой главе рассматриваются следующие вопросы.

- Разработка проекта отчета в визуальной среде Rave Reports.
- Компоненты `TRvProject` и `TRvSystem`.
- Просмотр и печать отчета.
- Использование в отчете внешних файлов.
- Преобразование форматов данных при помощи компонентов Rave Reports.

Этапы создания отчета и включение его в приложение

Процесс создания отчета с помощью генератора отчетов Rave Reports состоит из трех этапов. Первый выполняется в визуальной среде Rave Reports, второй и третий — в среде разработки Delphi.

1. На первом этапе в визуальной среде Rave Reports создается проект отчета и в нем необходимые страницы, объекты доступа к данным *(см. главу 32)*. На страницах располагаются элементы оформления, при необходимости к ним подключаются просмотры данных (если отчет отображает информацию из таблиц базы данных) и объекты аутентификации пользователей отчетов. Создаются обработчики событий. И в завершение этапа готовый проект отчета сохраняется в файле с расширением rav.
2. На втором этапе в проект приложения в Delphi переносятся компоненты `TRvProject` и `TRvSystem` *(см. главу 31)* из категории **Rave** Палитры инструментов. При этом в состав приложения автоматически включается ядро генератора отчета. Первый компонент связывается с файлом проекта отчета и представляет в приложении отчет со всеми его свойствами, страницами, элементами оформления и т. д. Второй компонент связывается с первым, взаимодействует с ядром генератора отчетов и обеспечивает печать отчета из приложения.
3. На третьем этапе создается программный код, обеспечивающий выполнение функций приложения, связанных с отчетом. Наряду со стандартными операциями предварительного просмотра и печати отчета это могут быть загрузка из файла и сохранение в файле, преобразование формата данных отчета, изменение содержания отчета в зависимости от выполненных пользователем действий и т. д. На этом этапе используются свойства, методы и методы-обработчики событий компонентов `TRvProject` и `TRvSystem` и других компонентов со страницы **Rave** Палитры компонентов.

Далее в этой главе мы подробно рассмотрим перечисленные этапы и различные аспекты программирования, связанные с ними.

Простой отчет в визуальной среде Rave Reports

При описании первого этапа наша задача — обсудить возможности проекта RAV, исследовать его структуру и составные части, которые доступны в приложении Delphi. Собственно, элементы оформления просты в примене-

нии, и мы акцентируем внимание лишь на нескольких, требующих небольших пояснений.

Для начала создадим в визуальной среде Rave Reports новый проект (команда **File | New** главного меню). Обратите внимание, что по умолчанию вместе с проектом создается первый отчет `Report1` с одной страницей `Page1`. Его мы и выберем, переименовав в `rptXP`. Проекту присвоим имя `SimpleDemo`.

Нумерация страниц отчета

В первую очередь создадим заголовок страницы и включим механизм нумерации страниц. Для этого перенесем на страницу элемент `Section` со страницы **Standard** Палитры инструментов и поместим его в верхней части страницы отчета. Эта секция будет объединять элементы оформления заголовка.

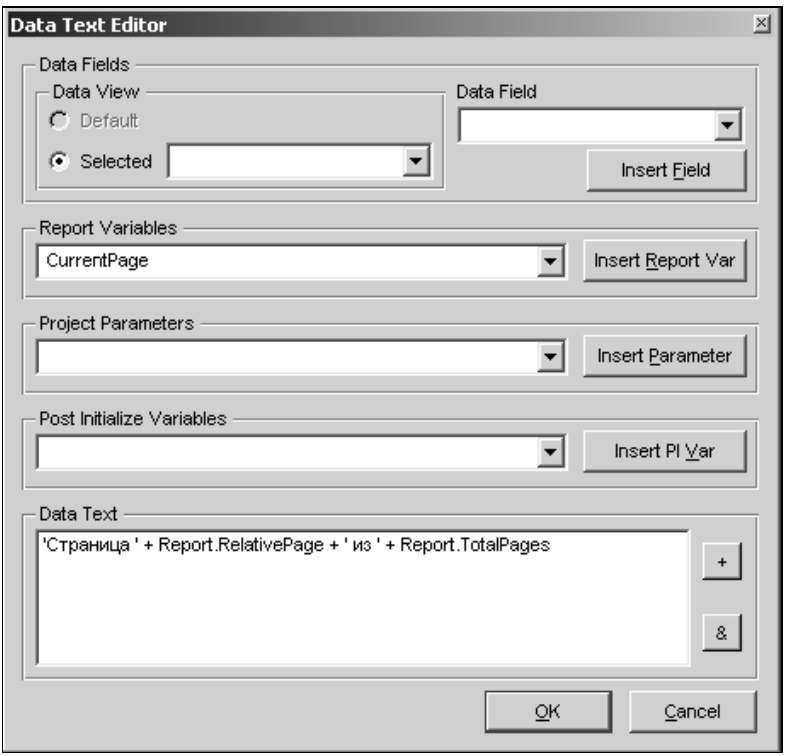


Рис. 33.1. Редактор свойства `DataField` элемента `DataText`

Поместим в секции элемент `DataText` — нам необходимо его свойство `DataField`. Редактор свойства **Data Text Editor** (рис. 33.1) обеспечивает та-

кую настройку, чтобы элемент мог отображать разнообразные данные. Сейчас нас интересуют номера страниц отчета.

Из списка **Report Variable**, который содержит глобальные переменные отчета, необходимо выбрать переменную **Relative Page**. Затем нужно щелкнуть на кнопке **Insert Report Var**, и переменная `Report.RelativePage` появится в поле **Data Text** в нижней части диалога. Эта переменная при печати отчета будет содержать порядковый номер текущей страницы.

Аналогичным образом добавим переменную **Total Pages**, которая возвращает общее число страниц отчета. Затем вручную отредактируем текст в поле **Data Text**:

```
'Страница ' + Report.RelativePage + ' из ' + Report.TotalPages
```

Шаблон номера страницы готов. Но для того, чтобы механизм нумерации заработал, необходимо перенести на страницу невизуальный элемент `PageNumInit`. Он будет работать автоматически. Единственное, что нужно сделать, так это настроить свойство `InitValue`, в котором задается номер начальной страницы.

Примечание

Обратите внимание, что элемент `PageNumInit` должен быть только один и располагаться на первой странице, которая должна быть пронумерована. Иначе нумерация каждой страницы, где есть такие элементы, начнется сначала.

Использование элемента *FontMaster*

Невизуальный элемент `FontMaster` позволяет установить одинаковые шрифты в элементах оформления, например, в пределах одной секции. Для этого надо перенести в секцию элемент `FontMaster` и задать в его свойстве `Font` необходимые характеристики шрифта. После этого во всех элементах, которые будут использовать этот шрифт, нужно в списке свойства `FontMirror` выбрать этот элемент `FontMaster`.

В результате один раз настроив шрифт, можно применить его для любого числа элементов оформления.

Добавление страниц к отчету

После создания первой страницы к отчету необходимо добавить еще две страницы и оформить их по образцу первой. Здесь все операции рутинны и не требуют дополнительных пояснений. В результате дерево проекта для отчета `rptXP` выглядит так, как показано на рис. 33.2.

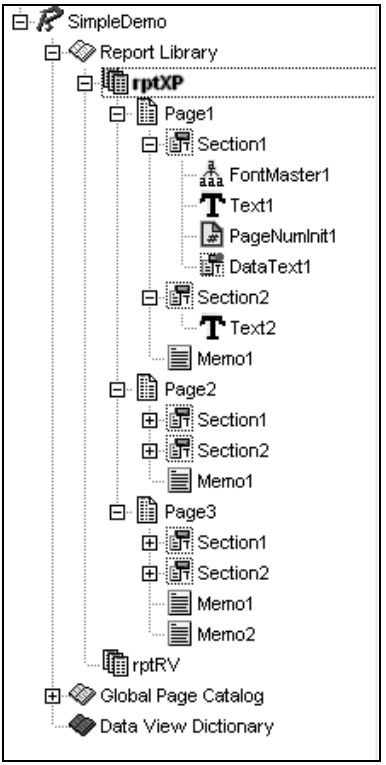


Рис. 33.2. Дерево проекта `SimpleDemo` для отчета `rptXP`

Кроме обычных страниц, принадлежащих отчету, в Rave Reports можно создавать глобальные страницы, которые можно связать с любым отчетом проекта (например, титульные страницы). Для создания новой глобальной страницы служит команда **File | New Global Page** главного меню. Затем процесс разработки не отличается от обычных страниц.

Теперь, для того чтобы при просмотре или печати отчета отображались все страницы, а не только первая, необходимо дополнительно настроить свойство `PageList` отчета. В редакторе свойства необходимо перенести в список **Page List** все нужные страницы. Для этого страница выбирается из выпадающего списка **Report Pages**. Затем нужно щелкнуть на кнопке **Add Page**. Аналогичная операция выполняется и для глобальных страниц проекта, доступных в списке **Global Pages**.

Результат для отчета `rptXP` представлен на рис. 33.3.

Кнопками слева в группе **Page List** можно изменять порядок следования страниц при печати отчета или удалять страницы из списка.

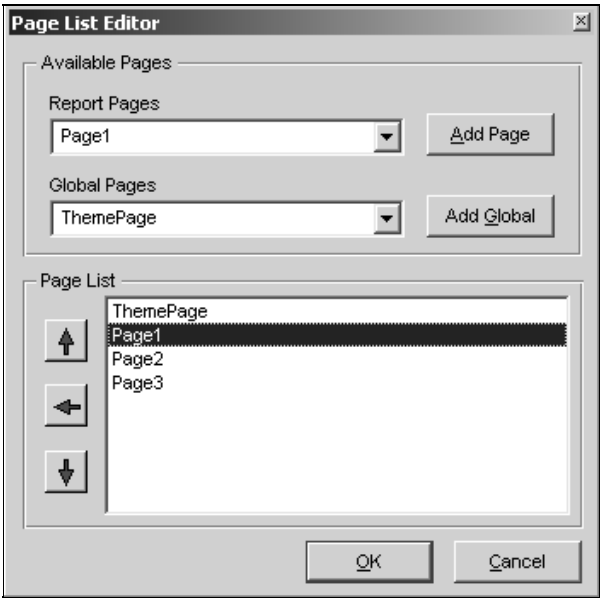


Рис. 33.3. Редактор свойства `PageList` отчета `rptXP`

Теперь отчет `rptXP` готов. Дополнительно к нему, и с помощью тех же элементов и действий, в рассматриваемом нами примере создан еще один отчет `rptRV`. При этом по умолчанию текущим считается отчет, который был текущим при последнем сохранении проекта.

Проект отчета сохранен в файле SimpleDemo.RAV.

Отчет в приложении

Теперь, когда проект SimpleDemo.RAV с двумя отчетами готов, перейдем к разработке приложения в Delphi.

Любое приложение, использующее генератор отчетов Rave Reports, должно иметь как минимум пару компонентов: `TRvProject` и `TRvSystem`. Первый из них есть проект отчета в приложении. С его помощью разработчик получает доступ к отчетам проекта и их свойствам. Второй компонент обеспечивает функционирование ядра генератора отчетов Rave Reports при печати или предварительном просмотре отчета (см. рис. 31.1). Подробнее о свойствах и методах этих компонентов *см. в главе 31.*

При реализации этих компонентов в состав исполняемого кода приложения автоматически включается ядро генератора отчетов Rave Reports. Соответственно при распространении приложения не требуются дополнительные файлы — даже файл проекта отчета можно включить в приложение.

Компонент `TRvProject` необходимо связать с файлом проекта SimpleDemo.RAV. Для этого имеется его свойство

```
property ProjectFile: string;
```

Файл RAV можно распространять вместе с приложением или включить его в состав исполняемого файла. Для этого существует свойство

```
property StoreRAV: boolean;
```

компонента `TRvProject` (см. рис. 31.2).

Перед использованием отчетов из компонента `TRvProject` его необходимо открыть. В нашем примере при открытии формы приложения компонент открывается и в список считываются полные имена отчетов проекта и отображается описание текущего отчета:

```
procedure TfmMain.FormShow(Sender: TObject);
begin
 rpProject.Open;
 rpProject.GetReportList(lbxRptList.Items, True);
 rpProject.ReportDescToMemo(meDesc);
end;
```

Но только один из отчетов доступен для работы одновременно. Для смены текущего отчета можно воспользоваться методом

```
function SelectReport(ReportName: string; FullName: boolean): boolean;
```

Для идентификации текущего отчета компонент `TRvProject` имеет два свойства, которые возвращают его имя и полное имя. Это, соответственно, свойства `ReportName` и `ReportFullName`. При необходимости задать имя отчета для одного из методов (например, `SelectReport`) можно указывать оба имени. Во всех методах, где в качестве параметра присутствует имя отчета, имеется дополнительный параметр `FullName` типа `Boolean`. Его значение `True` соответствует полному имени отчета.

Обратите внимание, что перед вызовом любых свойств и методов, относящихся к отчету в компоненте `TRvProject`, отчет необходимо открыть. Для этого предусмотрен метод `Open`:

```
RvProject1.Open;
```

или свойство `Active`:

```
RvProject1.Active := True;
```

Также, если в процессе работы приложения в компонент `TRvProject` нужно загрузить новый проект отчета, процедуру открытия нужно повторить:

```
...
RvProject1.Close;
RvProject1.SetProjectFile(dlgOpenProject.FileName);
```

```
RvProject1.Open;
...
```

Компонент отчета необходимо связать с компонентом `TRvSystem`. Для этого в свойстве `Engine` компонента `TRvProject` необходимо задать ссылку на компонент `TRvSystem`.

Просмотр и печать отчета

Если в предыдущем пункте, обсуждая отчет в приложении Delphi, мы говорили о компоненте `TRvProject`, то за выполнение любых операций с ним отвечает компонент `TRvSystem`.

При стандартной настройке этого компонента при печати или предварительном просмотре отчета всегда отображается диалог настройки печати (см. рис. 31.4). Если отображение этого диалога необходимо, печать текущего отчета компонента `TRvProject`, с которым связан данный компонент `TRvSystem`, осуществляется методом

```
procedure Execute;
```

любого из этих компонентов.

Если диалог настройки печати не нужен, компонент `TRvSystem` позволяет выполнить операцию напрямую. Для этого необходимо выполнить несколько действий.

Во-первых, нужно настроить свойство

```
type
  TReportDest = (rdPreview, rdPrinter, rdFile);property DefaultDest:
  TReportDest;
```

которое определяет, куда будет направлен отчет — в окно просмотра, на принтер или в файл.

Во-вторых, необходимо изменить свойство

```
type
  SystemSetup = (ssAllowSetup, ssAllowCopies, ssAllowCollate,
  ssAllowDuplex, ssAllowDestPreview, ssAllowDestPrinter, ssAllowDestFile,
  ssAllowPrinterSetup);  TSystemSetups = Set of TSystemSetup;property
  SystemSetups: TSystemSetups;
```

убрав из него опцию `ssAllowSetup`, которая включена по умолчанию:

```
RvSystem1.SystemSetups := RvSystem1.SystemSetups – [ssAllowSetup];
```

И наконец, свойству

```
property DoNativeOutput: Boolean;
```

необходимо присвоить значение `False`, т. к. по умолчанию оно имеет значение `True`, которое и заставляет компонент показывать диалог настройки печати перед выполнением операции.

Обратите внимание на очень важную деталь: чтобы все сделанные настройки действительно сработали, печать отчета необходимо выполнять только методом `Execute` компонента `TRvProject`!

Сохранение отчета во внешнем файле

При помощи методов компонента `TRvSystem` можно сохранить отчет для последующей печати в формате PRN или сохранить проект Rave Reports в формате RAV.

Для реализации первого варианта необходимо в качестве источника печати указать файл:

```
...
if dlgSavePRN.Execute then
 begin
  rsSystem.DoNativeOutput := False;
  rsSystem.DefaultDest := rdFile;
  rsSystem.SystemSetups := rsSystem.SystemSetups - [ssAllowSetup];
  rsSystem.OutputFileName := dlgSavePRN.FileName;
  rpProject.Execute;
 end;
...
```

Как видите, здесь мы воспользовались методикой прямой печати, описанной в предыдущем пункте, указав в качестве приемника файл с расширением prn, выбранный в стандартном диалоге выбора файла.

Если же нужно сохранить проект отчета в файле с расширением rav, можно использовать метод `SaveToFile` компонента `TRvProject`:

```
if dlgSaveProject.Execute
  then RvProject1.SaveToFile(dlgSaveProject.FileName);
```

Так же просто выполнить и обратную операцию — загрузить в компонент `TRvProject` проект отчета из файла, но при этом не забудьте закрыть текущий отчет:

```
...
RvProject1.Close;
RvProject1.LoadFromFile(dlgOpenProject.FileName);
RvProject1.Open;
...
```

Аналогичную функцию выполняет метод

```
procedure SetProjectFile(Value: String);
```

компонента `TRvProject`.

Компонент *TRvNDRWriter*

Компонент `TRvNDRWriter` предназначен для сохранения отчетов в файлах. При этом применяется двоичный формат NDR.

Приемник данных определяется свойством

```
type
  TStreamMode = (smMemory, smTempFile, smFile, smUser);property
  StreamMode: TStreamMode;
```

Здесь:

- `smMemory` — данные выводятся в поток в памяти (объект типа `TMemoryStream`);
- `smTempFile` — данные сохраняются во временном файле, в папке, определенной в операционной системе для хранения временных файлов;
- `smFile` — данные сохраняются в файле;
- `smUser` — данные передаются в поток, заданный разработчиком.

Имя файла, в котором будет сохранен отчет, определяется свойством

```
property FileName: String;
```

А для определения потока существует свойство

```
property Stream: TStream;
```

Таким образом, если вы хотите сохранить отчет в файл, перед использованием компонент настраивается, например, так:

```
...
RvNDRWriter1.StreamMode := smFile;
RvNDRWriter1.FileName := ReportFilePath;
...
```

Если вы планируете задействовать поток, сделайте так:

```
var ReportStream: TMemoryStream;
...
 ReportStream := TMemoryStream.Create;
 try
  RvNDRWriter1.StreamMode := smUser;
  RvNDRWriter1.Stream := ReportStream;
```

```
...
 finally
  ReportStream.Free;
 end;
...
```

Но сначала этот отчет необходимо создать. Для этого существует обширный набор методов, позволяющих отображать текст и графику, создавать таблицы и заполнять их данными. Перед началом работы необходимо вызвать метод

```
procedure Start;
```

а после окончания создания отчета — метод

```
procedure Finish;
```

Например, следующий фрагмент кода (листинг 33.1) создает в отчете текст с заданным положением.

Листинг 33.1. Пример текстового отчета с форматированием

```
with RvNDRWriter1 do
begin
 Start;
 Units := unMM;
 SetFont('Times New Roman', 14);
 Bold := True;
 OriginX := 0.0;
 OriginY := 0.0;
 GotoXY(1.0, 12.0);
 Print('Заголовок #1');
 GotoXY(6.0, 18.0);
 Println('Заголовок #2');
 GotoXY(6.0, 24.0);
 Println('Заголовок #3');
 GotoXY(6.0, 30.0);
 Println('Заголовок #4');
 Finish;
end;
```

А вот так можно нарисовать прямоугольник и разместить в нем изображение (листинг 33.2).

Листинг 33.2. Пример вывода прямоугольника с изображением

```
with RvNDRWriter1 do
begin
 Start;
 Units := unMM;
 SetBrush(clBlue, bsSolid, nil);
 Rectangle(5.0, 35.0, 65.0, 95.0);
 Bitmap := TBitmap.Create;
 Bitmap.LoadFromFile('factory.BMP');
 PrintBitmapRect(10.0, 40.0, 60.0, 90.0, Bitmap);
 Bitmap.Free;
 Finish;
end;
```

В данном фрагменте кода метод

```
procedure PrintBitmapRect(X1,Y1,X2,Y2: double; Bitmap: TBitmap);
```

отображает растровое изображение `Bitmap` в прямоугольнике, обеспечивая его масштабирование в соответствии с размерами прямоугольника.

Внимание

Компонент `TRvNDRWriter` имеет свойство `Canvas` *(см. главу 12)*, но использовать его нельзя — любые операции с канвой не возымеют действия, и ваши труды не будут сохранены.

При вызове пары методов `Start` и `Finish` не нужно предпринимать никаких дополнительных усилий для сохранения отчета — это будет сделано методом `Finish`.

Множество других свойств и методов компонента (мы не будем останавливаться на них специально, т. к. их использование достаточно прозрачно) обеспечивают оформление отчета, управление страницами, настройку принтера и т. д.

Преобразование форматов данных

С компонентом `TRvNDRWriter` (вернее, с файлами в формате NDR, которые он создает) взаимодействует ряд компонентов Rave Reports, которые обеспечивают преобразование данных из этого специфического формата в более распространенные.

Следующие компоненты обеспечивают преобразование отчета:

- `TRvRenderPDF` — в формат PDF для дальнейшего использования в Adobe Acrobat Reader;
- `TRvRenderHTML` — в формат HTML;
- `TRvRenderRTF` — в формат RTF;
- `TRvRenderText` — в текстовый формат.

Работа со всеми осуществляется по общей схеме.

Сначала необходимо загрузить отчет из файла NDR в поток.

После этого вызывается метод

```
procedure PrintRender(NDRStream: TStream; OutputFileName: TFileName);
```

который и выполняет преобразование:

```
var ReportStream: TMemoryStream;
...
 ReportStream := TMemoryStream.Create;
 try
  ReportStream.LoadFromFile(NDRFilePath);
  RvRenderHTML1.NDRStream := ReportStream;
  RvRenderHTML1.PrintRender(ReportStream, 'sdf');
 finally
  ReportStream.Free;
 end;
```

Резюме

Приложение Delphi, которое реализует печать отчетов, должно иметь в своем составе компоненты `TRvProject` и `TRvSystem`. Первый обеспечивает представление проекта отчета из файла RAV в приложении. Второй взаимодействует с ядром генератора отчетов и управляет печатью и предварительным просмотром отчетов.

Набор компонентов Rave Reports позволяет преобразовать формат данных отчета в наиболее распространенные форматы данных: HTML, RTF, PDF и TXT.

Глава 34

Отчеты для приложений баз данных

Генератор отчетов Rave Reports позволяет создавать отчеты, отображающие данные из различных источников данных. Для этого предназначены средства визуальной среды и компоненты из категории **Rave** Палитры инструментов Delphi.

Приложение может воспользоваться соединением, предоставленным генератором отчетов и созданным и настроенным в проекте отчета, а также компонентами Rave Reports, которые обеспечивают передачу данных в отчет.

При помощи компонентов и визуальной среды генератора отчетов Rave Reports приложение баз данных может работать с базами данных, обращаясь к ним при помощи различных технологий доступа к данным *(подробнее о технологиях доступа к данным см. часть V)*, доступных по умолчанию:

- ADO;
- dbExpress;
- BDE;
- InterBase.

Кроме этого, компонент `TRvDataSetConnection` (без участия визуальной среды) позволяет использовать любые наборы данных, открываемые через компоненты доступа к данным Delphi. Любой потомок компонента `TDataSet` также доступен в отчете Rave Reports. Это означает, что дополнительно к перечисленным технологиям отчеты Rave Reports могут задействовать клиентские наборы данных в распределенных приложениях *(см. часть VII)*.

Еще один интересный компонент — `TRvCustomConnection` — обеспечивает доступ к любым данным не из баз данных. Для него источником данных могут быть текстовые файлы, электронные таблицы, электронная почта и т. д.

Что касается элементов оформления отчетов, средствами визуальной среды Rave Reports можно создавать отчеты самых различных типов. Естественно, это простые табличные отчеты, а также отчеты, представляющие данные в

отношении "один ко многим", отчеты с группировкой данных и с вычисляемыми значениями.

В этой главе рассматриваются следующие вопросы.

- Подключение источника данных к отчету.
- Типы соединений с источниками данных, используемые в проекте отчета и приложении Delphi.
- Компоненты соединений с базами данных.
- Компонент `TRvCustomConnection`.
- Типы отчетов.
- Использование вычисляемых значений.

Соединения с источниками данных в Rave Reports

Если отчет Rave Reports должен отображать данные из какого-либо источника данных, на этапе разработки в визуальной среде в проект отчета должны быть добавлены специальные объекты, обеспечивающие соединение с источником данных и формирование набора данных, который затем отображается в отчете.

В Rave Reports существуют два типа соединений с источниками данных (рис. 34.1).

- Соединение через драйвер Rave Reports.
- Соединение через компонент Rave Reports и компонент набора данных в приложении Delphi.

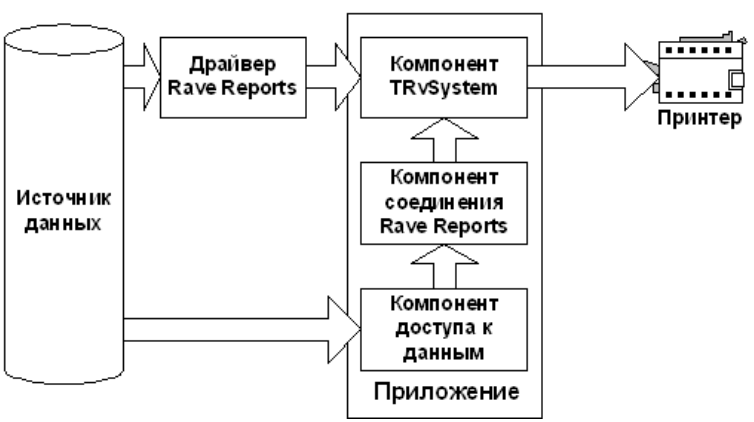


Рис. 34.1. Соединения с источниками данных в отчете Rave Reports

В обоих случаях соединение инкапсулировано в объекте визуальной среды, а различаются только способы доступа к данным и набор компонентов, необходимый для этого.

При соединении через драйвер Rave Reports проект отчета на этапе разработки и ядро генератора отчетов на этапе выполнения используют драйверы, которые реализованы в виде файлов с расширением rvd. Именно наличие этих файлов предопределяет выбор технологий доступа к данным при создании объекта соединения в среде разработки (рис. 34.2). В стандартную поставку Rave Reports включены драйверы для следующих технологий доступа к данным:

- ADO;
- dbExpress;
- BDE;
- InterBase Express.

При соединении через компоненты в Delphi сначала необходимо создать объект просмотра **Direct Data View**, который реализует прямой доступ к набору данных на основе активного соединения в приложении Delphi. При этом соединение может быть создано на основе любой доступной в Delphi технологии доступа к данным. Сделать это позволяет набор компонентов Rave Reports в категории **Rave** Палитры инструментов Delphi.

Реализованы следующие технологии доступа к данным:

- ADO;
- dbExpress;
- BDE;
- InterBase Express;
- клиентские наборы данных распределенных приложений;
- источники данных, не использующие базы данных (текстовые файлы, электронная почта, электронные таблицы и т. д.).

За визуализацию данных в отчете отвечают специализированные элементы оформления, расположенные на странице **Report** Палитры инструментов визуальной среды Rave Reports. Они связываются с полями объекта просмотра данных, созданного разработчиком. Это может быть просмотр на основе запроса SQL, созданного разработчиком, или прямой просмотр набора данных Delphi. Просмотр объединяет нужные поля из таблиц, которые доступны через соединение с источником данных *(см. главу 32)*. Это реализует свойство `DataView` элементов оформления.

Соединения с источниками данных в визуальной среде Rave Reports

Любой отчет, работающий с базами данных, должен быть настроен соответствующим образом в визуальной среде создания отчетов Rave Reports. Независимо от типа соединения здесь должен быть создан хотя бы один объект доступа к данным.

Если вы хотите создать соединение через драйверы Rave Reports, вам потребуется объект соединения и связанный с ним объект просмотра на основе запроса SQL.

При необходимости создать соединение на основе компонентов Delphi в визуальной среде вам потребуется объект прямого просмотра.

Рассмотрим подробнее действия, которые необходимо выполнить в визуальной среде Rave Reports для создания и настройки соединения.

Соединение через драйвер Rave Reports

Проект, отчеты которого используют соединение через драйверы Rave Reports, должен содержать два объекта доступа к данным (в дереве проекта они отображаются в ветви **Data View Dictionary**).

В первую очередь нужно создать объект соединения (компонент `Database`). Для этого служит команда **File | New Data Object** главного меню. В появившемся диалоге **Data Connections** (см. рис. 32.5) необходимо выбрать опцию **Database Connection** и нажать кнопку **Next** — появится список **Database Connection Type** (рис. 34.2). В нем представлены все доступные в визуальной среде типы соединений. Элементы списка соответствуют файлам RDV драйверов соединений. В стандартную поставку входят драйверы для соединений ADO, BDE, dbExpress и InterBase Express. Эти файлы по умолчанию устанавливаются в папке ..\Rave Reports\DataLinks.

После нажатия кнопки **Finish** появляется специализированный диалог настройки параметров соединения, типовой для выбранной технологии доступа к данным *(см. часть V)*, и создается новый объект соединения. В его свойствах `AuthDesign` и `AuthRun` содержатся настройки соединения, которые необходимы при его открытии на этапах разработки и выполнения отчета соответственно.

Теперь, когда соединение готово, необходимо создать просмотр данных. Он формирует набор данных, который будет передан в отчет и там связан с элементами оформления (см. рис. 34.1).

В диалоге выбора объекта доступа к данным (см. рис. 32.5) доступны два объекта просмотра. Но с соединением на основе драйвера Rave Reports

умеет взаимодействовать только один из них — **Driver Data View**. Для него необходимо выбрать соединение из созданных ранее в этом проекте (рис. 34.3).

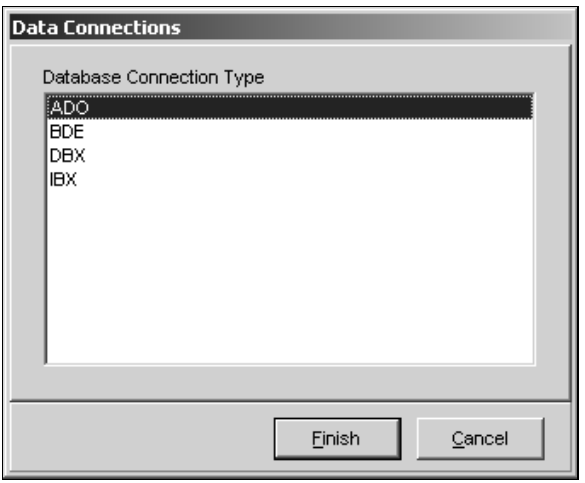


Рис. 34.2. Диалог выбора драйвера соединения

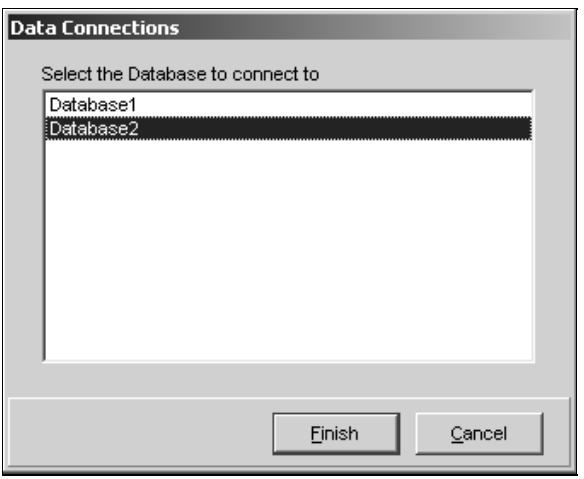


Рис. 34.3. Список выбора соединения для просмотра **Driver Data View**

После этого в редакторе **Query Advanced Designer** (см. рис. 32.6) создается запрос SQL, используемый для создания набора данных просмотра. Подробнее о пользовательском интерфейсе этого редактора рассказывается в *главе 32*.

После создания объекта просмотра (компонент `DriverDataView`) в его свойстве `Database` появляется ссылка на объект выбранного соединения, а свойство `Query` позволяет редактировать запрос SQL просмотра. Это обеспечивает все тот же редактор **Query Advanced Designer**. Соединение просмотра можно изменить, выбрав новое в списке свойства `Database`, однако при этом может потребоваться частично или полностью переписать запрос SQL.

В состав объекта просмотра входят объекты всех полей, определенных на этапе разработки для запроса SQL.

Теперь подготовленный в просмотре набор данных можно подключить к элементам оформления нужного отчета.

Соединение через компонент приложения Delphi

Второй тип соединения базируется на компоненте набора данных, который уже существует и соединен с источником данных в приложении Delphi. Основой такого соединения в приложении является один из специализированных компонентов из категории **Rave** Палитры инструментов Delphi — например, компонент `TRvDataSetConnection`. В проекте отчета в визуальной среде Rave Reports соединение обеспечивает объект просмотра **Direct Data View**. При создании для него необходимо выбрать из списка одно из соединений из приложения Delphi, в котором будет использоваться проект отчета. В этом списке можно выбрать все соединения, доступные как во время разработки, так и во время выполнения приложения. Для этого предусмотрены флажки в нижней части окна (рис. 34.4).

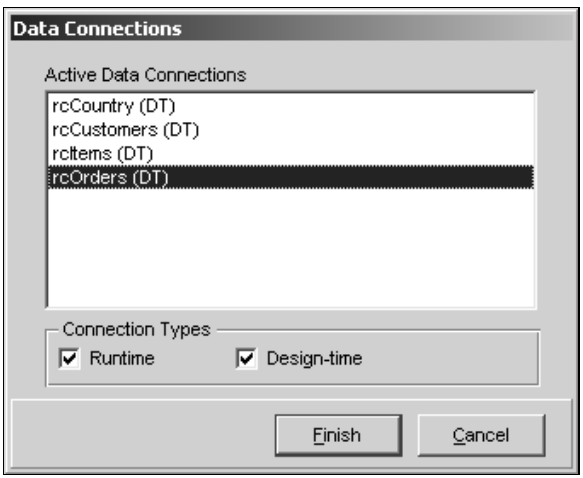


Рис. 34.4. Список доступных соединений из приложения Delphi

После создания объект прямого просмотра (компонент `DataView`) доступен в дереве проекта. В его состав входят все поля набора данных, связанного в приложении Delphi с компонентом соединения. И их можно подключить к элементам оформления нужного отчета.

Соединения с источниками данных в приложении

Теперь давайте посмотрим, как нужно работать со специализированными компонентами Rave Reports в приложениях Delphi для того, чтобы создать соединение отчета с источником данных. Их основная задача — передать в отчет связанный набор данных.

Для создания соединения можно выбрать следующие компоненты:

- `TRvCustomConnection` — обеспечивает доступ к источникам данных, не основанным на базах данных;
- `TRvDataSetConnection` — создает соединение с любыми компонентами наборов данных, предком которых является класс `TDataSet`;
- `TRvTableConnection` — создает соединение с компонентом `TTable`;
- `TRvQueryConnection` — создает соединение с компонентом `TQuery`.

После переноса на форму проекта Delphi эти компоненты становятся доступны для выбора при создании прямых просмотров **Direct Data View** в визуальной среде создания отчетов Rave Reports. Однако перед этим компонент соединения необходимо связать с набором данных.

Компонент *TRvDataSetConnection*

Компонент `TRvDataSetConnection` позволяет отчету получить доступ к наборам данных, инкапсулированных в любых компонентах, произошедших от класса `TDataSet`. Это открывает перед разработчиком самые широкие возможности по созданию отчетов для любых приложений баз данных и распределенных приложений.

Сразу после переноса на форму компонент становится доступным в визуальной среде Rave Reports при создании объекта прямого просмотра. Однако толк от ненастроенного соединения пока небольшой. Сначала его нужно связать с компонентом набора данных. Для этого предназначено свойство

```
property DataSet: TDataSet;
```

И это все. Теперь созданный в визуальной среде объект прямого просмотра автоматически получит объекты полей, соответствующие полям в наборе данных компонента `DataSet`. Впрочем, еще несколько вспомогательных свойств могут дать разработчику дополнительные удобства.

Примечание

Здесь мы рассмотрим только часть свойств и методов. Компонент `TRvDataSetConnection` обладает большой группой свойств и методов, которые, будучи использованы в методах-обработчиках событий, позволяют дополнительно оформлять отчет.

Свойство

```
property FieldAliasList: TStrings;
```

пригодится, если нужно изменить имена полей в прямом просмотре проекта отчета. Для этого в списке свойства в формате `Name=Alias` задаются имена полей связанного набора данных и их псевдонимы, которые будут использованы в объекте прямого просмотра.

Методы-обработчики событий компонента отслеживают процесс навигации по набору данных при печати отчета.

При открытии соединения для создания отчета генератором отчетов вызывается метод-обработчик

```
type
  TRPConnectorEvent = procedure(Connection: TRvCustomConnection);
property OnOpen: TRPConnectorEvent;
```

При открытии соединения отчет требует передать ему информацию о структуре набора данных (метаданные). Компонент соединения делает это и вызывает метод-обработчик

```
property OnGetCols: TRPConnectorEvent;
```

Когда курсор устанавливается на первую строку набора данных, вызывается метод-обработчик

```
property OnFirst: TRPConnectorEvent;
```

а при перемещении на следующую запись можно применить метод

```
property OnNext: TRPConnectorEvent;
```

Если генератор отчетов нашел нужную запись и считал ее для представления в отчете, для отслеживания этого события в распоряжении разработчика есть метод

```
property OnGetRow: TRPConnectorEvent;
```

При достижении последней записи набора вызывается метод-обработчик

```
type
  TRPEOFEvent = procedure(Connection: TRvCustomConnection; var Eof:
  Boolean);
property OnEOF: TRPEOFEvent;
```

При повторном использовании набора данных, например, при печати отчетов "один ко многим" или при группировке записей соединение обновляется. При этом все параметры соединения и набора данных приводятся к исходному состоянию и вызывается метод-обработчик

```
property OnRestore: TRPConnectorEvent;
```

Кроме этого, при печати сложных отчетов генератору отчетов может понадобиться некоторое подмножество отсортированных записей набора данных.

Перед началом сортировки и фильтрации и после их завершения вызываются пары методов-обработчиков

```
property OnGetSort: TRPConnectorEvent;
property OnSetSort: TRPConnectorEvent;
```

и

```
property OnGetFilter: TRPConnectorEvent;
property OnSetFilter: TRPConnectorEvent;
```

Однако вы можете обеспечить дополнительную фильтрацию записей, передаваемых из набора данных в отчет. Для этого существует метод-обработчик

```
type
  TRPValidateRowEvent = procedure(Connection: TRvCustomConnection;
  var ValidRow: Boolean);
property OnValidateRow: TRPValidateRowEvent;
```

Параметр `ValidRow` управляет отправкой отдельной записи отчету: при значении `True` запись пропускается в отчет.

Рассмотрим простой пример. Для отчета, печатающего всем нам хорошо известную таблицу COUNTRY из демонстрационной базы Delphi, можно ограничить список стран, и исключить страны с площадью территории, меньше заданной:

```
procedure TForm1.RvSomeConValidateRow( Connection: TRvCustomConnection;
var ValidRow: Boolean);
begin
 with TRvDataSetConnection(Connection) do
  ValidRow :=DataSet.FieldByName('Area').AsInteger > 1000000;
end;
```

Компоненты, использующие BDE

В состав набора компонентов соединений Rave Reports включены два компонента, которые обеспечивают связь прямого просмотра в проекте отчета с набором данных BDE.

Компонент `TRvTableConnection` работает с компонентом `TTable`. Для связывания с таблицей BDE имеется свойство

```
property Table: TTable;
```

Еще одно свойство

```
property UseSetRange: Boolean;
```

при значении `True` определяет, что при создании отчета будут задействованы механизмы фильтрации и сортировки компонента `TTable`.

Компонент `TRvQueryConnection` работает с компонентом `TQuery`.

Свойство

```
property Query: TQuery;
```

задает компонент `TQuery`.

Остальные свойства и методы этих компонентов соответствуют компоненту `TRvDataSetConnection`.

Компонент *TRvCustomConnection*

Компонент `TRvCustomConnection` обеспечивает доступ к самым разнообразным источникам данных. Фактически через этот компонент разработчик может передать в отчет все данные, какие только сможет загрузить в приложение. Причина столь удивительной универсальности кроется в том, что:

- во-первых, компонент изначально не ориентирован ни на один конкретный вид данных;
- во-вторых, работа по созданию строк отчета (посредством методов-обработчиков событий) возлагается на разработчика.

Повторим банальную истину, что чудес не бывает, и, как видите, за гибкость приходится расплачиваться дополнительным объемом работы.

Для того чтобы настроить соединение через компонент `TRvCustomConnection`, необходимо выполнить следующие действия.

1. Определить число строк отчета и установить его в компоненте.
2. Создать структуру данных отчета (метаданные). Здесь нужно решить, какие именно поля будут присутствовать в отчете, в каком порядке, дать им названия и определить их тип данных.

3. Создать процедуру, обеспечивающую передачу данных из источника данных в текущую строку отчета.
4. Связать компонент соединения с объектом прямого просмотра.

Обсудим эту последовательность действий более детально на простом примере. Создадим небольшое приложение, которое позволяет загружать текстовые файлы в два компонента `TMemo`. Перенесем на форму и настроим все необходимые компоненты Rave Reports.

Затем разработаем отчет, который печатает данные из этих двух компонентов в двух колонках. Отчет тоже несложен и состоит из полос заголовка и окончания и полосы данных с расположенными на ней двумя элементами оформления `DataText`.

Наша задача сейчас — настроить компонент `TRvCustomConnection` так, чтобы он мог отображать данные из двух компонентов `TMemo` (листинг 34.1).

Листинг 34.1. Методы-обработчики событий компонента `TRvCustomConnection`, обеспечивающего соединение отчета с массивами Memo

```
procedure TfmMain.rcCustomOpen(Connection: TRvCustomConnection);
begin
 Connection.DataRows := Max(meLeft.Lines.Count, meRight.Lines.Count);
 i := 0;
end;
procedure TfmMain.rcCustomGetCols(Connection: TRvCustomConnection);
begin
 Connection.WriteField('LeftColumn', dtString, 40, 'LeftColumn', '');
 Connection.WriteField('RightColumn', dtString, 40, 'RightColumn', '');
end;
procedure TfmMain.rcCustomGetRow(Connection: TRvCustomConnection);
begin
 if meLeft.Lines.Count >= i
  then Connection.WriteStrData('',  meLeft.Lines[i])
  else Connection. WriteNullData;
 if meRight.Lines.Count >= i
  then Connection.WriteStrData('',  meRight.Lines[i])
  else Connection. WriteNullData;
 Inc(i);
end;
```

При открытии соединения в методе-обработчике `OnOpen` рассчитывается число записей, необходимое для отображения наиболее длинного из двух файлов.

Метод-обработчик `OnGetCols` вызывается, когда отчету необходимы метаданные о наборе данных соединения. Здесь создаются два поля. Для этого служит метод

```
procedure WriteField(Name: String; DataType; TRPDataType; Width: Integer;
FullName: String; Description: String);
```

который создает поле в соответствии с переданными в нем параметрами.

И при печати отчета для каждой строки вызывается метод-обработчик `OnGetRow`, в котором задаются значения полей. Для каждого типа данных необходим свой метод:

```
function WriteBCDData(FormatData: String; NativeData: Currency):
String;function WriteBlobData(var: Buffer; Len: Longint): String;function
WriteBoolData(FormatData: String; NativeData: Boolean): String;function
WriteCurrData(FormatData: String; NativeData: Currency): String;function
WriteDateTime(FormatData: String; NativeData: TDateTime);function
WriteFloatData(FormatData: String; NativeData: Extended): String;function
WriteIntData(FormatData: String; NativeData: Integer): String;function
WriteNullData;function WriteStrData(FormatData: String; NativeData:
String): String;
```

Обратите внимание, что все эти методы не определяют, какому именно полю будет присвоено значение. Поэтому присваивание осуществляется в порядке следования полей: первый по порядку метод отправляет в отчет значение для первого поля, второй для второго и т. д.

Примечание

Методы-обработчики компонентов `TRvCustomConnection` и `TRvDataSetConnection` совпадают.

Теперь осталось связать соединение с проектом отчетов. Это делается стандартным образом — при создании объекта прямого просмотра. Но здесь есть одна особенность. Как уже говорилось, при создании прямого просмотра в нем автоматически создаются объекты полей, соответствующие полям набора данных. И теперь мы знаем, что у компонента соединения имеется специальный метод-обработчик `OnGetCols`, который вызывается при создании полей.

Однако если вы создадите объект прямого просмотра обычным способом, визуальная среда Rave Reports создаст один-единственный объект поля, не имеющего ничего общего с реальными метаданными. Для того чтобы поля импортировались в проект отчета правильно, необходимо, чтобы при создании объекта просмотра приложение, содержащее компонент `TRvDataSetConnection`, было

запущено. Тогда в диалоге выбора соединений (см. рис. 34.4) необходимо включить флажок **Runtime**, и вы увидите компонент нужного соединения. В этом случае объект прямого просмотра получит все необходимые поля, которые затем следует связать с элементами оформления отчета.

Аутентификация пользователя в отчете

Два объекта Rave Reports позволяют включить в проекте отчета механизм проверки имени пользователя и пароля. Это объекты **Simple Security Controller** (элемент `SimpleSecurity`) и **Data Lookup Security Controller** (элемент `LookupSecurity`), которые доступны для выбора в диалоге создания объектов доступа к данным визуальной среды Rave Reports (см. рис. 32.5). Создается новый объект командой **File | New Data Object** главного меню Rave Reports. Созданный объект появляется в ветви **Data View Dictionary** дерева проекта.

Элемент `SimpleSecurity` предназначен для хранения списка пользователей и их паролей. Он имеет свойство `UserList`, в котором в формате `UserName=Password` заносятся имена и пароли пользователей.

Элемент `LookupSecurity` обеспечивает загрузку имен пользователей и паролей из таблицы базы данных. Для этого к нему через свойство `DataView` должен быть подключен соответствующий просмотр данных. В свойстве `UserField` необходимо указать поле, которое содержит имена пользователей, а в свойстве `PasswordField` задается поле с паролями.

Теперь несколько слов о том, как подключить созданные объекты.

Вы можете организовать аутентификацию на двух уровнях: уровне проекта и уровне отчета. В обоих случаях используется свойство `SecurityControl` объекта проекта или отчета. В нем необходимо выбрать нужный объект аутентификации.

Однако это действие не увеличивает защищенность ваших отчетов, поскольку все необходимые проверочные операции придется написать самому в исходном коде приложения Delphi. Единственное отличие в аутентификации по уровням в том, где именно вы сможете получить доступ к объекту аутентификации — из компонента проекта или отчета.

Для организации простейшей проверки имени пользователя и пароля на уровне проекта в приложении, нужно написать примерно такой код:

```
...
rpReport.Open;
 if rpReport.ProjMan.SecurityControl.IsValidUser(
                                    edUserName.Text,
                                    edPassword.Text)
```

```
  then rpReport.Execute
  else ShowMessage('Доступ запрещен');
 rpReport.Close;
...
```

В данном случае доступ к объекту аутентификации `SecurityControl` (класс `TRaveBaseSecurity`) осуществляется через объект менеджера проекта `ProjMan` (класс `TRaveProjectManager`). Метод

```
function IsValidUser(AUserName: string; APassword: string): Boolean;
```

этого объекта возвращает `True`, если переданные в параметрах имя и пароль не совпадают со значениями из списка или базы данных.

Для уровня отчета код выглядит так:

```
...
rpReport.Open;
 if rpReport.ProjMan.ActiveReport.SecurityControl.IsValidUser(
                                    edUserName.Text,
                                    edPassword.Text)
  then rpReport.Execute
  else ShowMessage('Доступ запрещен');
 rpReport.Close;
...
```

Здесь объект `ActiveReport` (класс `TRaveReport`) представляет текущий отчет.

Типы отчетов

А сейчас мы займемся вопросами разработки собственно отчетов. Схема применения элементов оформления, работающих с объектами доступа к данным, стандартна для любых типов отчетов. Поэтому сначала мы рассмотрим общую методику на примере простого отчета, а затем перейдем к более сложным отчетам.

Для всех рассматриваемых типов отчетов создано демонстрационное приложение DemoReports.

Простой табличный отчет

Для создания отчетов по данным из источников, предоставленных объектами соединений и просмотров, существуют элементы оформления со страницы **Report** Палитры инструментов Rave Reports.

Основой, без которой нельзя использовать полосы (элементы `Band` и `DataBand`), является элемент `Region`. Он ограничивает часть страницы, на которой будут печататься данные.

Главную роль в отчетах для приложений баз данных играют полосы. Это невизуальные элементы оформления, моделирующие горизонтальную область или строку отчета. На странице **Report** доступны два таких элемента. Обычная полоса `Band` создает горизонтальную область, которая не изменяет свое абсолютное или относительное положение на странице. Например, созданная на основе элемента `Band`, полоса `TitleBand` всегда располагается в начале первой страницы отчета и оформляет заголовок таблицы (рис. 34.5). А полоса `FooterBand` будет напечатана сразу после основной таблицы — но ее конкретное положение на странице зависит от размера набора данных.

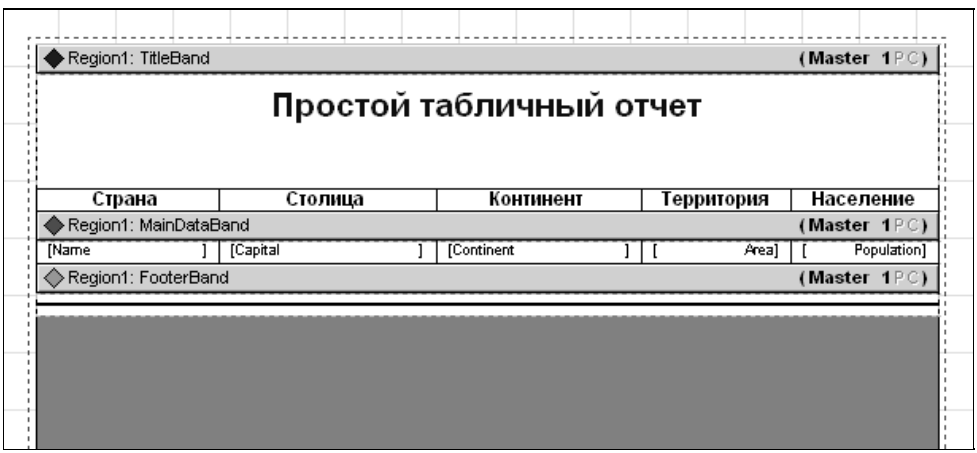


Рис. 34.5. Страница отчета SimpleReport в визуальной среде Rave Reports

Элемент оформления `DataBand` обеспечивает размножение строк отчета в соответствии с числом строк набора данных. Для этого полосу данных `MainDataView` необходимо связать с просмотром при помощи свойства `DataView`. В нашем примере это прямой просмотр `CountryView`, связанный с компонентом соединения `TRvDataSetConnection` в приложении.

На полосе данных `MainDataView` необходимо разместить элементы оформления `DataText`. Каждый из этих элементов связывается с объектом просмотра и полем данных этого просмотра. Для этого имеются свойства `DataView` и `DataField` соответственно. Таким образом, каждый элемент оформления `DataText` размножается вместе с полосой данных и формирует в отчете колонку значений поля набора данных.

Расположив на полосах горизонтальные и вертикальные линии, можно легко оформить данные в табличном виде.

Отчет "один ко многим"

При помощи средств Rave Reports можно создавать и более сложные отчеты. В приложениях баз данных очень часто встречаются отношения "один ко многим" между наборами данных.

Давайте посмотрим, как создать отчет "один ко многим". Само собой разумеется, он должен быть связан как минимум с двумя просмотрами, которые находятся в отношении "один ко многим".

> **Внимание**
>
> Для компонентов наборов данных в Delphi не нужно создавать отношение "один ко многим" — речь идет о том, что их поля позволяют такое отношение создать теоретически.

Подобно рассмотренному выше простому отчету, отчет "один ко многим" может содержать полосы `Band` и `DataBand`. Причем дополнительные настройки необходимы для обоих типов полос. Число полос `DataBand` должно соответствовать числу существующих в отчете наборов данных. Полосы `Band` несут в основном оформительскую нагрузку и их число зависит от эскиза отчета и вашей фантазии.

В качестве примера создадим отчет для двух таблиц из демонстрационной базы данных Delphi. Таблицы CUSTOMER и ORDERS находятся в отношении "один ко многим". Для них в тестовом приложении создано соединение с использованием ADO, и два табличных компонента ADO подключены к компонентам соединения `TRvDataSetConnection`.

Соответственно полоса данных `CustBand` будет отображать записи из набора данных `tCustomer`, а полоса `OrdBand` — из набора данных `tOrders` (рис. 34.6). Их необходимо связать с объектами прямых просмотров, как уже описывалось ранее для примера простого отчета.

А теперь займемся созданием отношения "один ко многим".

В подчиненной полосе данных `OrdBand` необходимо задать значения для четырех свойств.

1. Свойство `ControllerBand` должно содержать ссылку на главную полосу `CustBand`.
2. Свойство `MasterDataView` должно содержать ссылку на главный объект просмотра `CustomersView`.
3. В свойстве `MasterKey` необходимо задать ключевое поле `CustNo` главного просмотра `CustomersView`, по которому будет установлено отношение.

4. В свойстве `DetailKey` необходимо задать ключевое поле `CustNo` подчиненного просмотра `OrdersView`, по которому будет установлено отношение.

Рис. 34.6. Страница отчета MasterDetailReport в визуальной среде Rave Reports

Кроме этого, необходимо настроить атрибуты местоположения полос на странице отчета. Для этого предусмотрен Редактор полос отчета **Band Style Editor** (рис. 34.7), который открывается при щелчке по кнопке свойства `BandStyle` Инспектора объектов визуальной среды Rave Reports. В нем в группе **Print Location** для подчиненной полосы `OrdBand` необходимо установить флажок **Detail**.

На этом настройка отношения "один ко многим" завершена. Однако скажем еще несколько слов об оформлении такого рода отчетов с помощью обычных полос. В нашем примере две дополнительные полосы `OrdHeaderBand` и `OrdFooterBand` помогают визуально выделить группы записей подчиненной полосы. Для этого необходимо в их свойстве `ControllerBand` выбрать полосу данных `OrdBand`. Затем в редакторе полос отчета в группе **Print Location** для полосы `OrdHeaderBand` необходимо выбрать флажок **Body Header**, а для полосы `OrdFooterBand` — флажок **Body Footer**.

Обратите внимание (рис. 34.6 и 34.7), что значки маркировки на полосах на странице и в редакторе полос наглядно демонстрируют текущий статус полосы. Цветом выделены уровни вложения данных и подчиненность полос. Полосы с маркировкой одного цвета печатаются в одном блоке. Квадраты обозначают полосы данных, а треугольники — обычные полосы, при этом направление вершины треугольника обозначает полосу заголовка или окончания. А левая панель редактора полос отчета **Band Style Editor** (рис. 34.7)

наглядно демонстрирует модель отчета, как если бы он был напечатан для трех записей набора данных.

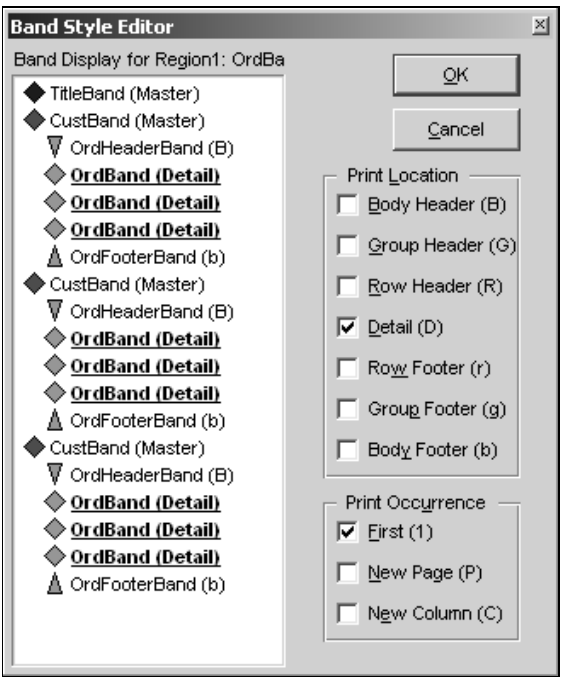


Рис. 34.7. Редактор полос отчета **Band Style Editor** для отчета MasterDetailReport

На основе отчета "один ко многим" можно легко разработать и более сложные. Для этого необходимо детальную полосу данных связать с новыми детальными полосами и настроить по описанной методике отношение "один ко многим".

Группирующий отчет

Отчеты, работающие с базами данных, часто должны отображать данные с различными уровнями группировки. Обычно группировка осуществляется в наборе данных, если он создается на основе запроса SQL с применением оператора `GROUP BY`. Но в самом наборе данных невозможно предусмотреть оформление групп записей, однако это можно сделать в отчете.

Для полосы данных, которая отображает данные из просмотра с группировкой, можно создать полосы группового заголовка и группового окончания. Для этого в наличии имеются свойства `GroupDataView` и `GroupKey`.

Первое должно указывать на объект группирующего просмотра, а второе задает поле или несколько полей, по которым осуществляется группировка. Применительно к оформлению отчета это означает, что при изменении значения группового ключа будут напечатаны полосы группового заголовка и окончания.

В качестве таких полос могут быть обычные полосы и полосы данных. Обычные полосы хороши, если группировка имеет один уровень вложенности (для каждого значения группового ключа существует одна или несколько сгруппированных записей). Полосы данных применяются, если группировка имеет несколько уровней (внутри группы выделяется еще один групповой ключ и каждая запись в группе имеет еще несколько сгруппированных записей второго уровня).

Кроме этого, для полос группового заголовка необходимо в свойстве `ControllerBand` задать основную полосу данных и настроить свойство `BandStyle`. Для группового заголовка в редакторе **Band Style Editor** в группе **Print Location** устанавливается флажок **Group Header**, а для полосы группового окончания — флажок **Group Footer**.

Обычно группирующие запросы SQL используют агрегатные функции для вычисления одного или нескольких величин по всей группе. Чаще всего это общая денежная сумма или общее количество. Такие величины удобно размещать в полосах группового окончания.

Использование вычисляемых значений

На странице **Reports** Палитры инструментов визуальной среды Rave Reports доступны несколько компонентов, которые позволяют применять агрегатные функции к значениям полей набора данных, переданного через соединение в отчет.

К агрегатным относятся следующие функции:

- AVG — вычисление среднего;
- COUNT — подсчет числа элементов множества;
- MAX — нахождение максимального значения;
- MIN — нахождение минимального значения;
- SUM — суммирование.

Простые элементы оформления позволяют получать вычисляемые значения на базе полей одного источника данных, более сложные умеют объединять несколько источников данных.

Рассмотрим все эти элементы оформления.

Вычисляемые значения по одному источнику

Для вычисления агрегатного значения одного или нескольких полей одного источника данных есть два элемента оформления.

Элемент `CalcText` позволяет отобразить результат вычисления на полосе отчета. Так же как и обычный элемент `DataText`, его необходимо связать с просмотром и полем. Для этого предназначены свойства `DataView` и `DataField` соответственно.

Кроме этого, элемент `CalcText` должен быть связан со специализированным элементом `CalcController` (см. далее), который будет управлять процессом вычисления. Связывание осуществляется при помощи свойства `Controller`.

В свойстве `CalcType` задается одна из пяти перечисленных агрегатных функций.

Дополнительно для функции COUNT можно настроить еще три свойства. При необходимости включить или отключить подсчет нулевых значений или пробелов применяются булевские свойства `CountNulls` и `CountBlanks` соответственно. А свойство `CountValue` позволяет задать значение поля, которое будет учитываться при расчете функции — все остальные значения будут игнорироваться.

Вычисленное значение при сохранении может быть отформатировано в соответствии с шаблоном, заданным свойством `DisplayFormat`.

Для всех агрегатных функций можно задать момент начала вычислений. Для этого в свойстве `Initializer` необходимо указать элемент оформления отчета, и при его печати начнется вычисление. Это может быть любой элемент, расположенный с элементом `CalcText` на одной полосе, но лучше — специализированный элемент `CalcController`.

Примечание

Пример использования элемента оформления `CalcText` имеется в отчетах, рассмотренных нами ранее.

Элемент `CalcTotal` является невизуальным аналогом элемента `CalcText`. Поэтому он обладает всеми свойствами, описанными для элемента `CalcText`. Вычисленное при его печати значение разработчик может использовать по своему усмотрению после того как оно сохранено. Приемником вычисленного значения может быть один из параметров объекта отчета.

Свойство `DestParam` позволяет выбрать один из предопределенных или созданных разработчиком параметров отчета (свойство `Parameters` объекта отчета).

Свойство `DestPiVar` задает переменную отчета, в которую будет передано вычисленное значение (свойство `PIVars` объекта отчета).

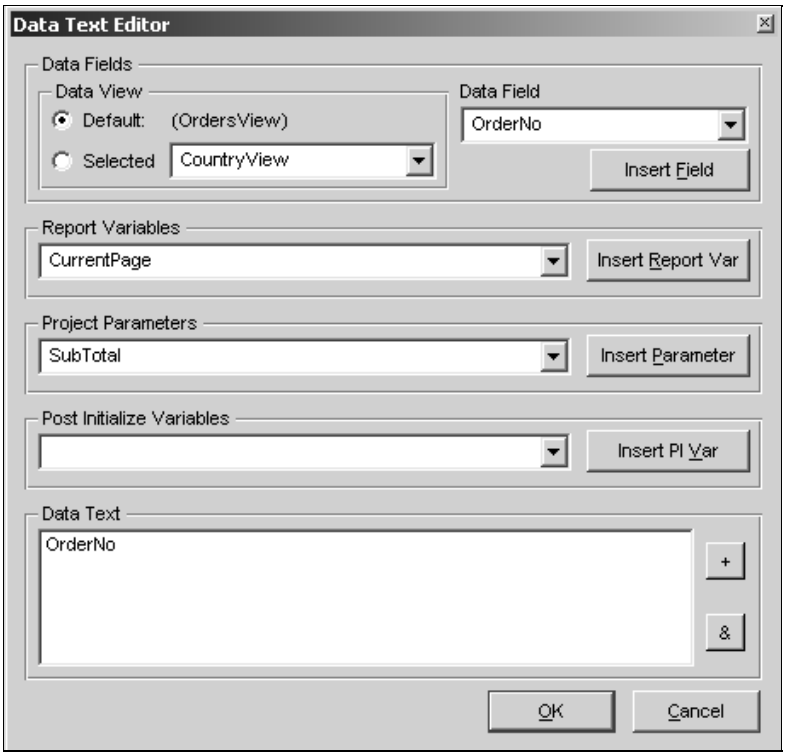


Рис. 34.8. Редактор свойства `DataField`

Затем параметр или переменная могут быть использованы для дальнейших вычислений или напечатаны при помощи элемента `DataText`. В редакторе свойства `DataField` этого элемента (рис. 34.8) параметры и переменные отчета можно выбрать из списков **Project Parameters** и **Post Initialize Variables**.

Вычисляемые значения по нескольким источникам

Вычислительный элемент `CalcOp` позволяет проводить вычислительные операции над значениями из двух различных источников.

Разработчик должен задать исходные значения и источники данных с помощью двух наборов свойств (табл. 34.1). Назначение части этих свойств вам уже знакомо.

Таблица 34.1. *Свойства элемента* `CalcOp` *для определения двух источников данных*

Первый источник	Второй источник	Назначение свойства
`Src1CalcVar`	`Src2CalcVar`	Определяет вычисляемый элемент, результат которого берется в качестве исходного
`Src1DataField`	`Src2DataField`	Задает поле просмотра, над значениями которого проводятся вычисления. Игнорируется при заданном свойстве `SrcXCalcVar`
`Src1DataView`	`Src2DataView`	Задает просмотр, над значениями поля которого проводятся вычисления. Игнорируется при заданном свойстве `SrcXCalcVar`
`Src1Function`	`Src2Function`	Позволяет выбрать математическую функцию (их список гораздо шире, чем просто агрегатные функции), которая будет выполнена над исходным значением перед вычислением основной операции элемента
`Src1Value`	`Src2Value`	Задает фиксированное значение, над которым производится вычислительная операция

Собственно функция, которая должна обработать значения из двух заданных источников, задается свойством `Operator`.

После выполнения основной операции результат может быть обработан еще один раз, если вы зададите математическую функцию в свойстве `ResultFunction`.

Таким образом, при помощи элемента `CalcOp` разработчик может реализовывать довольно сложные вычисления.

Если задать в качестве двух источников данных;

- два фиксированных значения (свойства `Src1Value` и `Src2Value`);
- два поля из одного или двух просмотров данных (свойства `Src1DataField` и `Src2DataField`);
- комбинацию первых двух вариантов,

то их значения будут последовательно обработаны вычислительной операцией, которую вы зададите свойствами

- `Src1Function`
- `Src1Function`
- `Operator`
- `ResultFunction`

Кроме этого, элемент `CalcOp` позволяет создавать вычислительные цепочки, если указать в качестве одного или двух источников другие вычислительные элементы (рис. 34.9).

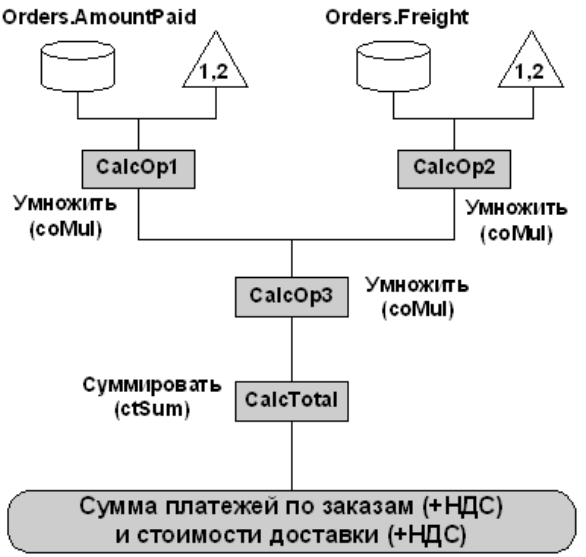


Рис. 34.9. Пример вычислительной цепочки на основе элементов `CalcOp`

Это могут быть как простые элементы `CalcText` и `CalcTotal`, так и другие элементы `CalcOp`, которые, в свою очередь, могут содержать сколь угодно сложные вычислительные цепи.

Пример элемента `CalcOp` имеется в демонстрационном приложении DemoReports на компакт-диске, прилагаемом к этой книге.

Управляющие вычислительные элементы

Ранее мы упоминали о свойстве `Controller`, элементов `CalcText` и `CalcTotal`, которое позволяет определить момент начала вычислений. Для этого существует специальный невизуальный элемент `CalcController`. Обычно он располагается на той же полосе, что и вычислительные элементы, и инициализирует процесс вычисления в момент своей печати. Хотя на самом деле невизуальный элемент `CalcController` не печатается, тем не менее, событие `OnBeforePrint` он получает исправно вместе со всеми элементами, расположенными на данной полосе. А значит и с инициализацией вычислений он справится вполне.

Обладая несколькими специфическими свойствами, он позволяет определить момент начала вычислений более точно. И так же, как элемент `FontMaster`, применяется для централизованного управления шрифтами, этот элемент может быть центром управления вычислениями.

Свойство `InitCalcVar` должно ссылаться на другой вычислительный элемент. И вычисленное им значение будет использовано в качестве начального.

Свойство `InitDataField` задает поле данных, значение которого является начальным. Работает, если свойство `InitCalcVar` не задано.

Свойство `InitValue` задает начальное значение, если предыдущие два свойства не заданы.

Для того чтобы эти свойства работали и задавали начальное значение, ссылка на элемент `CalcController` должна присутствовать в свойстве `Initializer` элементов оформления `CalcText` или `CalcTotal`.

Элемент `DataCycle` необходим для дополнительной фильтрации, сортировки и просмотра данных. С его помощью можно получить нужное для вычислений подмножество записей набора данных, не изменяя просмотра данных.

Свойство `DataView` задает просмотр данных, с которым будет работать элемент `DataCycle`.

При помощи свойств `MasterDataView`, `MasterKey` и `DetailKey` можно получить подмножество записей для отношения "один ко многим".

Свойство `SortKey` позволяет отсортировать записи по заданному полю.

Резюме

Генератор отчетов Rave Reports позволяет создавать разнообразные отчеты для приложений баз данных.

Совокупность компонентов Delphi и средств визуальной среды Rave Reports обеспечивает создание соединений с источниками данных любых видов, в том числе и не основанных на СУБД. Соединения могут применять технологии доступа к данным, реализованные соответствующими компонентами Delphi *(см. часть IV)* или драйверами Rave Reports.

Основой отчетов являются элементы `Region`, `Band` и `DataBand`, которые обеспечивают размножение строк отчета в соответствии с записями источников данных. Набор специализированных элементов оформления позволяет отображать в отчетах все основные типы данных.

ЧАСТЬ IX

Основы разработки Internet-приложений

Глава 35

Введение в архитектуру сетей

Роль транспортной составляющей в распределенных приложениях весьма велика. Без средств, соединяющих разрозненные части в единое целое, существование распределенных систем невозможно. Локальные сети, удаленный доступ через Internet, intranet — все это можно считать составной частью процесса разработки распределенных приложений.

Вам не обязательно быть сетевым инженером для разработки распределенных приложений, но элементарное понимание сетевых концепций, несомненно, окажется полезным. В Delphi предусмотрена реализация огромного набора сетевых протоколов, и чтобы остановить свой выбор на нужном варианте, следует знать их назначение и способы взаимодействия друг с другом, а в случае возникновения ошибки уметь грамотно локализовать ее.

Перед тем как начать перечислять протоколы и их свойства, рассмотрим принципы упорядочивания, позволяющие представить их как единую и непротиворечивую систему. Такая иерархия есть и называется Моделью взаимодействия открытых систем. Если вы с первых слов ждете названий функций и классов, то вам придется потерпеть — прежде нужно усвоить основные теоретические постулаты.

Полный анализ этой сложной системы не является целью данной главы. Желающие узнать больше о реализации модели OSI в Windows могут обратиться к документации Resource Kit.

В этой главе рассматриваются следующие вопросы.

- Семь уровней модели взаимодействия открытых систем (OSI).
- Физический уровень.
- Канальный уровень.
- Сетевой уровень.
- Транспортный уровень.
- Сессионный уровень.
- Уровень представления.
- Уровень приложения.

Модель OSI

"На заре туманной юности" сетей каждый производитель разрабатывал свои, закрытые решения. Очень скоро стало ясно, что это противоречит самой природе сетей — ведь они предназначены для того, чтобы объединять, а не разобщать. В 1978 г. Международная организация по стандартизации (International Standards Organization, ISO) ввела Модель взаимодействия открытых систем (Open Systems Interconnection, OSI), которая объединяет протоколы связи в семиуровневую структуру. Каждый уровень решает небольшую часть общей задачи связи. Уровни нумеруются, начиная с 1 (нижнего) и заканчивая 7.

Модель OSI распространяется только на множество протоколов связи, но не на физическую среду связи. Иногда говорят о модели OSI, как о *стеке протоколов*, поскольку каждому уровню, находящемуся выше, предшествует уровень, представляющий собой отличный от других, законченный, функционально полный набор протоколов, предназначенный для решения определенных задач в сфере коммуникаций.

В соответствии с идеологией OSI роль каждого уровня — предоставлять определенный интерфейс протоколам следующего уровня. Уровни абстрагированы друг от друга; протокол, скажем, четвертого (транспортного) уровня полагает, что "разговаривает" на понятном ему языке с тем же уровнем на другом компьютере. Фактически же он обращается к нижележащему уровню, тот — еще ниже и т. д. (рис. 35.1).

Для связи между собой компьютеры должны иметь функционально одинаковые стеки протоколов, которые обеспечивают связывающимся друг с другом компьютерам возможность общения, несмотря на то, что эти компьютеры могут иметь разную архитектуру. Например, компьютеры на основе процессоров фирмы Motorola (компьютеры Macintosh) хранят байты в слове, начиная со старшего. Компьютеры же с архитектурой "Wintel" (Windows + Intel) помещают первым младший байт в слова. "Просто так" Macintosh и PC не могут общаться друг с другом из-за этого простого, но важного отличия в способе хранения данных. Именно для этого и нужны стеки коммуникационных протоколов. Областью действия стеков является трансляция специфичных форматов данных разных компьютеров на общий язык, который будет понятен любым компьютерам, связанным при помощи стека протоколов.

Компьютер Macintosh, использующий стек TCP/IP, может напрямую связываться с PC, который также работает со стеком TCP/IP. Но замена на одном из компьютеров стека TCP/IP на стек IPX/SPX разорвет связь. В этом случае для трансляции протоколов вам необходим специальный маршрутизатор.

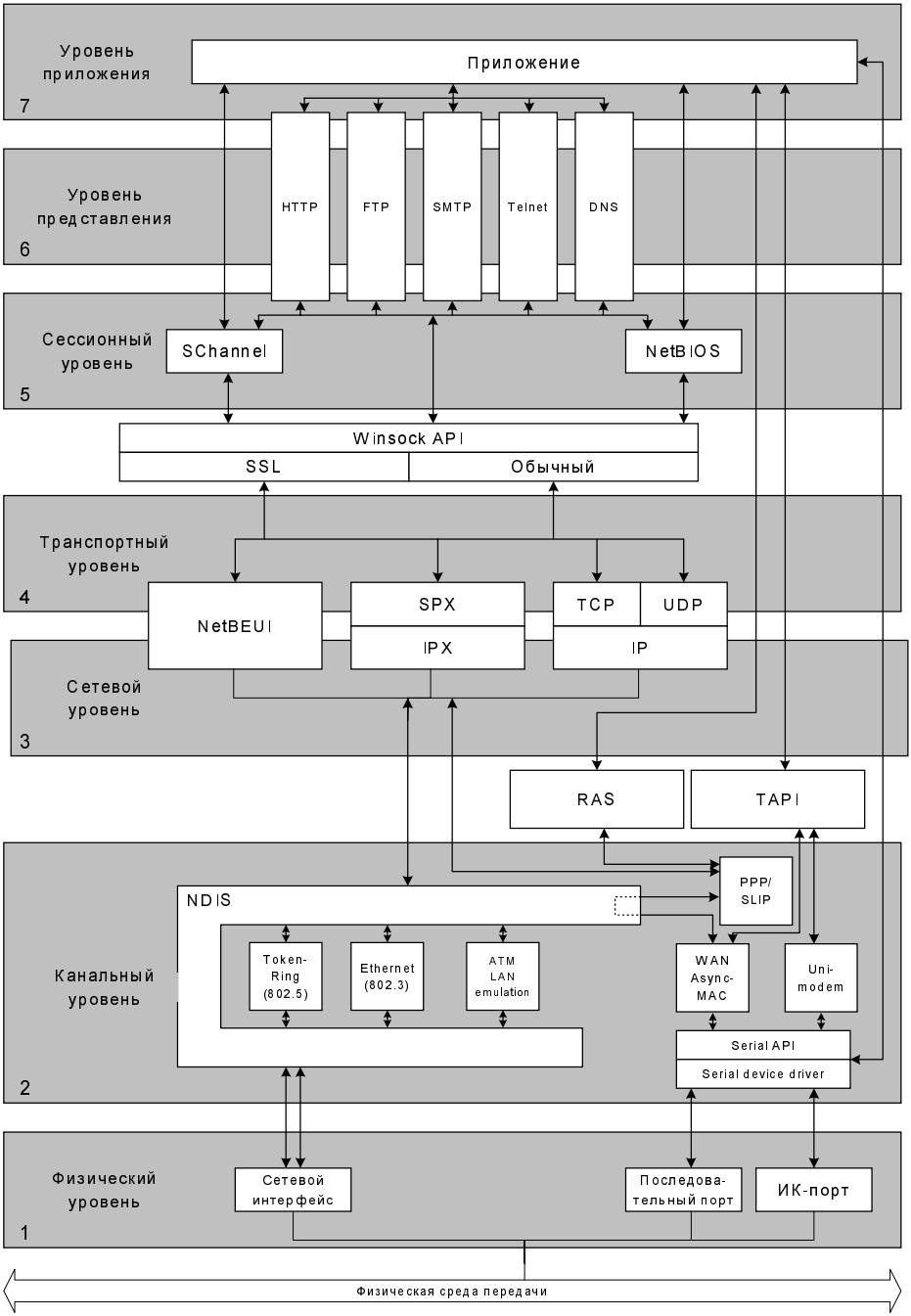


Рис. 35.1. Взаимодействие между уровнями модели OSI

Далее дается обзор уровней модели OSI. Он вряд ли сделает вас специалистом по сетям, однако поможет понять определенные вещи, необходимые для создания распределенных приложений.

Физический уровень

Физический уровень модели OSI непосредственно взаимодействует с сетевой передающей средой — электрическим или оптическим сетевым кабелем, телефонной сетью, устройствами связи по радио и т. п.

На физическом уровне еще нет структурированных данных — блоков, пакетов, записей. Все, за что он отвечает, — это преобразование последовательности бит в электромагнитный сигнал при передаче и обратное преобразование — при приеме.

Информация на физическом уровне передается в виде как цифровых, так и аналоговых сигналов. Компьютеры часто преобразуют сигналы из одной формы в другую. Одна из разновидностей процесса преобразования цифровой информации в аналоговую форму называется *модуляцией*. Обратное преобразование аналоговой информации в цифровую форму, которую понимает компьютер, называется *демодуляцией*. От этих двух терминов произошло название МОДЕМ (МОдулятор/ДЕМодулятор) — типичный пример устройства, обеспечивающего передачу цифровой информации по аналоговым (телефонным) каналам связи.

Аналоговые сигналы изменяются непрерывно и могут принимать произвольное множество значений. Цифровые сигналы изменяются дискретно. Двоичный сигнал принимает состояние 0 либо 1. К примеру, ISDN-адаптеры — это устройства цифровой связи, которые позволяют компьютерам соединяться между собой без применения модуляции/демодуляции.

Поскольку в реальной сети физическая среда может использоваться двумя и более устройствами одновременно, на этом уровне решается вопрос о топологии и совместном доступе к этой среде. Большинство из вас так или иначе, соприкасались с сетями стандарта Ethernet (мы еще рассмотрим его подробнее); в нем предусмотрена как шинная топология (множество устройств, соединенных общим коаксиальным кабелем), так и звездообразная (каждое устройство соединяется с концентратором (hub) или коммутатором (switch) индивидуальным сетевым кабелем).

Как правило, все, что нужно знать пользователю о физическом уровне, — это как правильно присоединить сетевой кабель к адаптеру. Здесь все реализуется аппаратно, программные интерфейсы и протоколы отсутствуют.

Протоколы канального уровня

Этот уровень решает следующие задачи сетевого соединения:

- идентификацию устройств в сети;
- контроль доступа в сетевую среду путем определения подходящего момента начала передачи;
- установление соединения между двумя физическими устройствами;
- управление потоком данных — последовательные прием и передача фрагментов данных;
- подтверждение приема фрагмента, инициализация повторной передачи неподтвержденных фрагментов, разрешение коллизий;
- контроль и коррекция ошибок физического уровня.

После принятия стандарта OSI физический и канальный уровни подверглись дальнейшей детализации. Эта детализация была проделана в рамках так называемого Проекта 802, выполненного Институтом инженеров в области электротехники и электроники (Institute of Electrical and Electronics Engineers, IEEE). С тех пор эти стандарты так и называются IEEE 802, хотя впоследствии были утверждены американскими и международными организациями стандартизации. Цифры 802 означают год и месяц начала работы над стандартом (февраль 1980 г.). Дату его принятия можно считать днем рождения современных локальных сетей. Изначально спецификация состояла из 10 разделов (802.1—802.10); раздел 802.3 стал основой сетей типа EtherNet, 802.4 — ArcNet, 802.5 — Token Ring.

Канальный уровень, согласно IEEE 802, состоит из двух подуровней.

- Подуровень *управления доступом к среде (Media Access Control — MAC)* определяет, каким образом множество устройств совместно используют один и тот же канал среды.
- Подуровень *управления логическим каналом (Logical Link Control — LLC)* устанавливает и поддерживает связь между соединяющимися устройствами.

Управление доступом к среде

Уровень MAC определяет, каким образом сигналы перемещаются в среде. Примерами могут служить соревнование за право передачи (Ethernet), пересылка маркера по кольцу (Token Ring) или шине (ArcNet) и голосование (мэйнфреймы).

Стоит отметить, что существует уникальный адрес, связанный с каждым сетевым устройством, например, с платой Ethernet вашего компьютера. Такой адрес называется физическим адресом устройства или MAC-адресом; он однозначно определяет некоторое аппаратное устройство в сети.

Если вас интересует MAC-адрес сетевой карты на вашем собственном компьютере, запустите утилиту `ipconfig` (для ОС семейства Windows NT/2000/XP) или `winipcfg` (для Windows 9*x*).

Запустите также из командной строки утилиту `arp` с параметром `-a`; эта утилита входит в состав всех современных версий Windows (при условии, что установлен протокол TCP/IP). Она покажет вам состояние кэша протокола ARP (Address Resolution Protocol), который отвечает за отображение IP-адресов на MAC-адреса компьютеров в вашей локальной сети.

В то время как протоколы физического уровня имеют дело с побитовой передачей данных, уровень управления линией передачи данных оперирует битами, объединенными в группы — так называемые фреймы.

Еще одно, что следует знать об уровне MAC, — это принципы синхронизации передачи данных. Существует три типа синхронизации передачи, осуществляющих управление этими группами:

- асинхронный;
- синхронный;
- изохронный.

Асинхронная передача. Классический пример асинхронной передачи — это соединение модема с модемом. Каждый из них использует стартовый бит, позволяющий принимающему его устройству произвести настройку на начало сигнала. Типовой асинхронный фрейм состоит из четырех компонентов:

- стартовый бит;
- биты данных;
- бит четности;
- бит останова.

Стартовый бит и бит останова сигнализируют о начале и окончании каждого фрейма. Биты данных несут информацию, которая посылается принимающему устройству. Бит четности служит для простейшего определения наличия ошибки. Устройство может определить ошибку, исказившую один бит во фрейме, но не ошибку, влияющую на два или более битов.

Синхронная передача. Синхронная связь более эффективна, чем асинхронная, поскольку время приема и передачи данных обеими сторонами синхронизируется. Этого можно достичь двумя путями:

- передавать синхронизирующий сигнал в потоке данных;
- использовать отдельный канал связи для передачи сигналов, несущих информацию о времени.

Синхронизация по времени на обоих концах канала связи позволяет передавать огромные фреймы данных, поэтому простой проверки четности уже недостаточно. При синхронизации каналов связи применяется циклическая проверка избыточности (Cyclic Redundancy Check, CRC) текущих данных.

Синхронная передача характерна, например, для большинства типов локальных сетей.

Изохронная передача. Изохронная передача подразумевает общее устройство, которое посылает временной сигнал всем устройствам, связанным с ним. Поскольку компьютеры PC — это независимые элементы, снабженные своими собственными таймерами, то встреча с передачей такого типа крайне маловероятна.

Где с рассмотренными понятиями может столкнуться программист? На уровне MAC работают все драйверы сетевых адаптеров. Вряд ли вы открыли эту книгу в поисках информации о написании драйверов (для этого существует Driver Development Kit), так что перейдем к следующему подуровню.

Управление логическим каналом (Logical Link Control)

Как упоминалось ранее, на этом подуровне осуществляется и обслуживается соединение устройств; он описан в стандарте IEEE 802.2. Здесь обеспечиваются управление потоком, контроль над ошибками и последовательностью передачи.

Как называется компонент Windows, работающий на канальном уровне? API LLC-уровня именуется NDIS (Network Driver Interface Specification). В соответствии с этой спецификацией пишутся все драйверы сетевых адаптеров и других оконечных устройств. Причем NDIS является общей базой, объединяющей как сетевые устройства, ориентированные на соединение (connection-oriented; к таким относятся сети ATM и ISDN), так и устройства без установления соединения (connectionless; в эту категорию попадают сети Ethernet, Token Ring и т. п.).

Помимо передачи данных, NDIS выполняет и другие сервисные функции, например, Plug-and-Play (автоматическое распознавание) или управление питанием. Если вы конфигурируете ваш компьютер таким образом, чтобы он "просыпался" от наступления сетевого события, то знайте, что этот сигнал будет подан посредством NDIS.

Помимо NDIS канальный уровень в Windows представлен еще одним протоколом — PPP (Point-to-Point Protocol). PPP служит для удаленного доступа, как правило, по последовательному порту; физическим устройством выступает в этом случае модем.

Примечание

Клиенты удаленного доступа Windows для совместимости поддерживают и более старый протокол SLIP (Serial Line Internet Protocol) — прародитель PPP.

Функции сетевого уровня

В то время как канальный уровень обеспечивает связь между физическими устройствами, соединяющимися в рамках одной сети, сетевой уровень реализует логику, позволяющую соединяться устройствам, находящимся в физически раздельных сетях. Этот уровень является основой межсетевого общения.

Поскольку межсетевое общение чрезвычайно распространено и в нем применяется различная сетевая аппаратура, то сетевой уровень содержит алгоритмы маршрутизации для доставки пакетов от источника к месту назначения.

Для сетевого уровня характерен такой термин, как *дейтаграмма (datagram)*. Дейтаграмма — это отдельный, независимый от других пакет данных, содержащий все необходимое для доставки информации от отправителя к адресату.

Примечание

Далее наряду с понятием дейтаграммы мы будем употреблять термин "пакет". Отличия между ними есть, но они достаточно сложны, чтобы быть упомянутыми в этой книге. Кроме того, на физическом и канальном уровнях принято оперировать термином "кадр" (frame).

Все устройства, участвующие в межсетевом общении, должны иметь для обеспечения маршрутизации свой собственный сетевой адрес. Все потенциальные адресаты в глобальной сети называются *хостами*. Проще всего думать о хосте как о компьютере; для простой рабочей станции с одним сетевым адаптером это в целом справедливо. Однако есть еще и серверы, которые могут быть оснащены несколькими сетевыми интерфейсами, и специализированные устройства (маршрутизаторы), которые если и являются компьютерами, то весьма специализированными.

В различных протоколах задача назначения адреса решается по-разному; естественно, мы рассмотрим подробно только протокол сетевого уровня IP (Internet Protocol), как основу для доставки информации в Сети.

Каждый хост в IP имеет уникальный параметр — *адрес*. По адресу дейтаграммы отправляются в определенную сеть и определенному хосту, участвующему в межсетевом общении. Естественно, каждый адрес даже в рамках Internet уникален — за этим следит специальная организация.

Наверняка подавляющее большинство читателей, так или иначе, соприкасалось с IP-адресацией, поэтому мы не будем здесь останавливаться на очевидных вещах. Напомним лишь, что IP-адрес, состоящий из четырех чисел в диапазоне от 0 до 255, содержит не только уникальный адрес хоста, но и уникальный адрес сети, в которой он находится. В зависимости от класса сети он может занимать один, два или три байта из четырех.

Если вы посмотрите спецификацию протокола IP в официальных стандартах, то увидите там такое определение: IP is a connectionless, unreliable datagram protocol primarily responsible for addressing and routing packets between hosts (IP — это протокол без установления соединения и проверки достоверности данных, отвечающий за адресацию и маршрутизацию пакетов между хостами). Что означают с виду непонятные термины *без установления соединения* (*connectionless*) и *ненадежный* (*unreliable*)?

Существуют три типа протоколов:

- протоколы, ориентированные на соединение (connection-oriented);
- протоколы без установления соединения и без подтверждения (unacknowledged connectionless);
- протоколы без установления соединения с подтверждением (acknowledged connectionless).

Протоколы, ориентированные на соединение (connection-oriented), похожи на телефонную связь. Они функционируют по хорошо отработанному маршруту, который существует до тех пор, пока есть соединение.

Такой протокол обеспечивает управление потоком и контроль над ошибками и последовательностью передачи, используя подтверждение приема каждой порции данных (передается специальный встречный пакет — подтверждение). Подобный двунаправленный обмен снижает пропускную способность, но повышает надежность.

Протоколы без установления соединения и без подтверждения (unacknowledged connectionless) заранее предполагают надежность сетевого соединения, например, в случае локальных сетей. Поэтому они не занимаются управлением потоком данных и обнаружением ошибок, оставляя эти функции протоколам других уровней. Отсутствие подтверждения приема дейтаграмм компенсируется быстротой работы.

Пример из "аналогового" мира? Пожалуйста: обычная почта. Почтовые службы развитых стран достигли надежности, пожалуй, 99,99%, и вы можете быть уверены в доставке каждого письма. Еще одно сходство с протоколом IP в том, что почтовая служба передает сообщение не напрямую, а от одного узла к последующим до тех пор, пока оно не достигнет получателя.

Однако сотню или более лет назад надежность почты была значительно ниже, надежность Internet пока также не везде близка к идеалу. Воспользуйтесь утилитами `ping` или `pathping` (входит в состав Windows 2000) и проверьте надежность доставки пакетов к выбранному вами хосту. Если сервер адресата не присоединен к Сети надежным каналом с хорошей пропускной способностью, вы можете потерять вплоть до 100% пакетов.

Протоколы без установления соединения с подтверждением (acknowledged connectionless) используют подтверждение о получении данных для управления потоком, но они не требуют предварительного установления соединения. Эти протоколы отличаются от рассмотренных ранее так же, как обычные письма от заказных: при доставке последнего вы получите соответствующее уведомление.

Так что ничего страшного в "ненадежности" IP нет. Этот недостаток будет скомпенсирован другими протоколами.

Итак, как следует из приведенного определения, главная задача IP — это маршрутизация. Каждый хост поддерживает собственную таблицу маршрутизации, которая содержит следующую информацию:

- список известных сетей и хостов и маски, определяющие принадлежность IP-адреса к данной сети;
- шлюз (gateway), через который можно попасть в данную сеть;
- сетевой интерфейс, с которого следует отправлять данные в эту сеть;
- метрику — число промежуточных соединений (hops), требуемых для попадания в сеть назначения по этому маршруту. Метрику также называют стоимостью (cost) маршрута. Если в нужную сеть ведут сразу несколько маршрутов, то выбирается тот, который имеет наименьшую стоимость.

Если вы настраивали протокол TCP/IP на вашем компьютере, то наверняка обращали внимание на такой параметр, как шлюз по умолчанию (default gateway). В локальных сетях таблицы маршрутизации компьютеров настроены таким образом, что пакеты, адресованные соседним компьютерам из той же сети, отправляются им напрямую. А шлюзу по умолчанию отправляются все пакеты, чья сеть недоступна отправителю напрямую.

Но для отсылки данных в локальной сети кто-то должен подсказать протоколу IP соответствие между IP- и MAC-адресами хостов. Этим занимается специальный вспомогательный протокол ARP (Address Resolution Protocol), входящий в состав стандарта IP.

Перед тем как перейти на уровень выше, подчеркнем: протокол сетевого уровня IP абсолютно достаточен, чтобы отправить данные из любой точки, где есть соединение с Internet, в любую другую точку. Проблемы соедине-

ния, за которые обычно отвечают сетевые администраторы, как правило, сводятся к неправильной настройке маршрутизации на этом важном уровне.

Транспортный уровень

Известное всем название TCP/IP наводит на мысль, что в Internet связь обеспечивают два протокола — сетевой IP и транспортный TCP (Transmission Control Protocol). Но это не совсем так. Помимо TCP, на транспортном уровне есть еще и протокол UDP (User Datagram Protocol). Тут мы переходим от чисто теоретических вопросов к прикладным. С обоими протоколами вы можете встретиться уже в конкретных компонентах Delphi. Разница между ними в том, что TCP является протоколом, ориентированным на соединение (connection-oriented), и логично дополняет IP средствами обеспечения надежности. UDP — минимальная надстройка над IP, не устанавливающая соединения. Этот протокол используется в специфических случаях, когда нужна скорость, а потеря некоторых порций данных допустима, скажем, при потоковой передаче видео- или аудиопотоков. Еще один пример — игра Quake, где UDP служит для координации игроков в сетевой игре.

Задачу протокола TCP можно представить себе как сборку головоломок (puzzles). На стороне отправителя происходит разборка исходной "картинки" на фрагменты. При чтении предыдущих разделов главы задумывались ли вы над таким важным показателем, как размер передаваемого фрагмента данных? Разумеется, методы передачи 10 Кбайт и 10 Мбайт должны отличаться. TCP может работать с сообщениями любой длины, принимая их от протоколов более высоких уровней, "нарезая" передаваемое сообщение на части и передавая их с IP-пакетами.

Примечание

Размер фрагментов составляет единицы-десятки килобайт и настраивается в зависимости от характеристик передающей среды.

Как нам уже известно, далее при путешествии по сети эти пакеты могут потеряться, быть задержаны, украдены, повреждены, продублированы; наконец — и это совершенно нормально — они могут прийти к получателю не в том порядке, в каком были отправлены. На приемной стороне TCP бережно проверяет пакеты и собирает их в правильном порядке. Верификация пакетов производится посредством проверки их контрольных сумм. При получении "правильного" пакета получатель оповещает отправителя сигналом подтверждения (ACK, от Acknowledgement), в противном случае передача повторяется.

Этот уровень также отвечает за процедуры, обеспечивающие надежную доставку сообщений. Протоколы нижних уровней рапортуют о возникающих ошибках, а транспортный уровень анализирует эти ошибки и либо требует повторной передачи данных, либо сообщает более высокоуровневым протоколам о необходимости предпринять соответствующие действия.

Если протокол сетевого уровня IP занимается доставкой сообщений от хоста к хосту, то транспортный уровень передает сообщения между процессами, запущенными на сетевых устройствах. Поэтому к IP-адресу на транспортном уровне добавляется еще один параметр соединения — *номер порта*.

По номеру порта дейтаграммы отправляются определенным процессам, запущенным на хосте-адресате. Если хост сравнить с радиостанцией, то порт — это частота, на которой она вступает в связь с другой радиостанцией. Операционная система, работающая в многозадачном режиме, может обслуживать сразу несколько сетевых приложений. Когда дейтаграмма приходит на компьютер, номер порта подсказывает операционной системе, какая из задач является получателем пришедших данных. Номер порта задается числом в диапазоне от 0 до 65 535.

Номер порта часто называется не иначе, как *хорошо известным портом (Well-known port)*. В стандартах Internet определяется несколько хорошо известных портов для таких протоколов, как: HTTP (порт 80), FTP (20, 21), telnet (23) и др. Под стандартные порты зарезервированы номера вплоть до 1023. Хорошо известные порты определены в модуле Winsock.PAS как константы, они начинаются с префикса `IPPORT_` (к примеру, `IPPORT_FINGER`).

Можно использовать для стандартного протокола порт, отличный от хорошо известного. Однако о таком изменении должны знать обе стороны, устанавливающие соединение. Хорошо известный порт потому так и называется, что дает клиенту возможность найти адресата без дополнительных уведомлений.

Концепции сессионного уровня, уровней представления и приложений

Хорошо известные протоколы, сервисы и приложения Internet находятся на трех верхних уровнях ссылочной модели OSI. Разделение между ними достаточно условное. Так, например, в описании TCP/IP все, что лежит поверх транспортного уровня, называется "уровнем приложений". Действительно, адресат определен, соединение установлено, целостность данных обеспечена, ошибки устраняются. Что еще остается на верхних уровнях? Только разговаривать в свое удовольствие на понятном друг другу языке.

Далее дан краткий обзор функций этих уровней.

Сессионный (session) уровень

Сессионный уровень обеспечивает собственно управление диалогом двух взаимодействующих устройств. Каждый акт, когда две взаимодействующие стороны устанавливают соединение, обмениваются данными и затем по уведомлению завершают его, будем называть сеансом (сессией). В ходе сеанса происходит взаимная идентификация сторон, определение параметров предстоящего сеанса (эту процедуру часто называют "рукопожатием" (handshaking)) и типа обмена сообщениями (однонаправленный, двунаправленный синхронный или асинхронный).

Уровень представления

На уровне представления решается задача приведения форматов данных "к единому знаменателю".

Когда связь осуществляется на низких уровнях, протоколы передают информацию побитно. Уровень представления определяет единый для всех взаимодействующих сторон порядок преобразования данных. Здесь могут выполняться:

- преобразование вида кодировки символов;
- преобразование форматов данных (целочисленных, с плавающей точкой, форматирование строк и т. п.);
- сжатие (компрессия) данных для уменьшения их объема;
- шифрование (для предотвращения несанкционированного прочтения или подмены *(более подробную информацию об этом см. в главе 36)*.

Уровень приложения

Наконец, к этому моменту наши многострадальные участники сетевого соединения устранили все реальные и потенциальные проблемы. Теперь они могут приступать к тому, для чего собственно и соединялись — решать ту или иную задачу.

Вы, безусловно, слышали о таких "языках": протокол передачи файлов (File Transfer Protocol, FTP), простой протокол передачи почты (Simple Mail Transfer Protocol, SMTP) и протокол эмуляции терминала (Terminal Emulation, telnet) — все они располагаются на этом уровне.

Еще раз подчеркнем, что разделение на уровни сессионный/представления/приложения достаточно условно. Реальные протоколы несут в себе черты всех трех. Возьмем, к примеру, всем известный протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP). Он включает в себя механизм организации и идентификации сеансов, авторизацию пользователей,

определение формата, в котором клиент посылает запросы HTTP-серверу, и выбор собственно языка этих запросов (GET/POST и т. п.). HTTP — фундамент браузера, его уровень приложения. С другой стороны, для протокола SOAP (на котором базируются Web-службы) HTTP — это не более чем сеансовый протокол, средство для обмена данными собственного, более высокого уровня.

Резюме

Архитектура сетей строится на модели взаимодействия открытых систем (OSI), которая подразумевает разделение на семь уровней. На каждом уровне реализованы стеки протоколов. В соответствии с идеологией OSI, роль каждого уровня — предоставлять определенный интерфейс протоколам следующего уровня. Уровни абстрагированы друг от друга. Благодаря модели OSI компьютеры с различной архитектурой могут понимать друг друга и общаться по сети.

Глава 36

Криптографическая защита информации в Internet

За последнее десятилетие Internet перешел в зрелый возраст, и из средства общения энтузиастов и фанатиков стал ареной для ведения бизнеса. А серьезный бизнес невозможен без соблюдения строгих формальных процедур, призванных минимизировать риски и потери.

Сеть Internet по сути является децентрализованной и неуправляемой (хорошо, минимально управляемой), и помимо бизнес-партнеров в ней вы можете повстречать кого угодно, включая разного рода злоумышленников. Вот почему любые важные данные, пересылаемые по Сети, должны быть надежно защищены от нежелательного прочтения, порчи или подмены.

Имя науки, занимающейся этими проблемами, — *криптография*. Из оружия в руках противоборствующих спецслужб криптография стала инструментом, необходимым буквально каждому пользователю. Конечно, технические детали защищенных систем по возможности скрыты от дилетантов. Но разработчику знание принципов и методов криптографии просто необходимо. Сотни программистов пытаются самостоятельно придумать и воплотить собственные "шифры", "кодеры", "системы ограничения доступа", не догадываясь, что станут жертвой взлома в течение считанных часов, если не минут. И в такой ситуации находятся не только самодеятельные разработчики — пароли для Windows 98, старых версий Microsoft Office, популярных архиваторов вскрываются за столь же короткое время. Вот почему браться за решение таких проблем следует только после основательного знакомства с современной криптографией.

В этой главе мы сначала рассмотрим круг прикладных задач криптографии. Затем познакомимся с понятиями цифровой подписи и сертификата. Далее, на примере менеджера сертификатов изучим особенности интерфейса `CryptoAPI`. И закончим вопросами настройки Web-средств и компонентов, используемых в распределенных приложениях для криптографической защиты.

Основные термины и понятия криптографии

Начинающие программисты часто понимают под криптографией тривиальное шифрование и расшифровку данных. Да, это первая и очень важная задача с тысячелетней историей, однако сферы применения криптографии значительно шире, в нее входят:

- обеспечение секретности (защита от несанкционированного чтения информации) (privacy);
- аутентификация (проверка подлинности) информации и идентификация законных пользователей системы;
- контроль целостности информации.

Секретность

Секретность (privacy) обеспечивается при помощи алгоритмов шифрования (encryption). Определение секретности звучит очень просто — передаваемые данные не должны быть прочитаны за разумный промежуток времени никем, даже если они были полностью перехвачены. Собственно, для передачи данных по сети Internet последнее является скорее правилом, чем исключением. За время "путешествия" между двумя удаленными компьютерами пакеты IP могут проходить десятки хостов, и заранее определить и тем более задать их путь нельзя.

Любой алгоритм шифрования — это правило, отображающее исходное содержимое сообщения (часто называемое *исходным текстом, plaintext*) на *зашифрованную последовательность данных (ciphertext)*. Хотя в определении встречается слово "текст", это — не более чем дань традиции. Шифруемые данные могут быть любыми — содержимым базы данных, аудио- или видеофайлом, исполняемым кодом.

Данные шифруются при помощи ключа — особой последовательности данных, которая, будучи по специальному алгоритму "смешана" с исходными данными, обеспечивает их изменение до такой степени, чтобы декодирование без второго ключа было невозможным. Разумеется, любое шифрование является обратимой операцией — имея ключ, можно осуществить операцию дешифрирования, в результате которого данные будут однозначно восстановлены (рис. 36.1).

Внимательные читатели заметили, что было сказано "без второго ключа", а не "без такого же ключа". В самом деле, закрыв дверь вашей квартиры на все замки, вы можете быть уверены, что ее сможет открыть лишь обладатель другого набора таких же, как у вас ключей (разумеется, если дверь обладает достаточной стойкостью, и ее не сможет быстро открыть злоумышленник с помощью отмычек и кувалды).

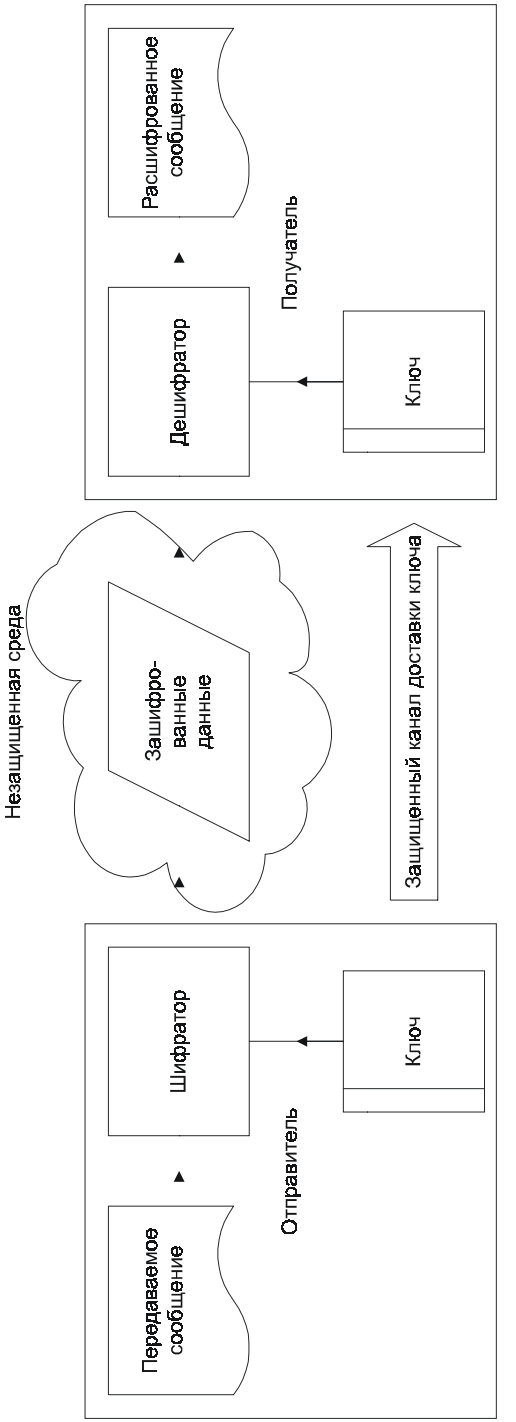


Рис. 36.1. Схема организации доставки секретной информации

К счастью, в цифровой криптографии дело обстоит не так. Есть алгоритмы, в которых ключи шифрования и дешифрирования совпадают — они называются *симметричным шифрованием*. Единый для двух операций ключ в этом случае называется секретным. Он известен только передающей и приемной сторонам. Любой посторонний, узнавший ключ, получает, во-первых, возможность прочитать зашифрованную информацию, во-вторых, — и это зачастую не менее важно — он получает возможность шифровать и отправлять ложную информацию, выдавая себя за подлинного отправителя.

В 1976 г. была опубликована работа Диффи и Хэллмана (W. Diffie, M. E. Hellman), которая открыла эпоху *асимметричного шифрования, или шифрования с открытым ключом*. Существуют способы создания ключевых пар, один из ключей в которой — открытый, или публичный — применяется для шифрования. Второй — закрытый, или приватный — для расшифровки. Только его владелец может прочитать секретное сообщение. Знают его даже не два (как в первом случае), а один человек. Такой ключ никуда не надо передавать или транспортировать, что повышает надежность системы.

Математически несложно развернуть эту схему на 180 градусов — шифрование осуществляется секретным ключом, а расшифровка — открытым. Что это за секретность, спросите вы, ведь расшифровать и прочитать сообщение сможет кто угодно? Верно, но это не совсем шифрование, а... вы вероятно узнали идею электронной цифровой подписи. Если она вам еще не знакома, то мы очень скоро обсудим ее более подробно. Таким образом, вокруг с виду простой идеи о двух различных ключах построено все здание современной криптографии.

Как ни парадоксально это звучит для неподготовленного пользователя, во всем мире отдают предпочтение открытым и известным алгоритмам шифрования. Дело в том, что широкомасштабные эксперименты всего криптографического сообщества позволяют "всем миром" выявить и устранить потенциальные "дырки" и слабости алгоритма. А секретный алгоритм, пусть и добавит труда аналитикам противодействующей стороны, но зато при обнаружении дефектов сразу становится уязвимым и бесполезным. Кроме того, рано или поздно все тайное становится явным, и строить политику секретности на сокрытии алгоритмов несерьезно. Еще в XIX в. голландский ученый Керкхофф сформулировал правило: криптостойкость шифрования должна сохраняться даже в том случае, если оппоненту известны все механизмы и исходные данные, за исключением секретного ключа. Ключ — это всего лишь параметр системы, и даже при единичной утере его относительно легко заменить, не меняя всю систему в целом.

Не нужно быть математиком, чтобы понять, что ваш оппонент, обладающий бесконечными вычислительными ресурсами и бесконечным временем, рано или поздно расшифрует любое сообщение конечной длины, хотя бы путем простого перебора вариантов. Изюминка здесь как раз в том, чтобы приме-

нить достаточно стойкий алгоритм, делающий расшифровку путем перебора бессмысленной. Действительно, любая тайна, будь то коммерческая, государственная или любовная, рано или поздно устаревает. И если нет уверенности в достижении цели за время, пока эта тайна актуальна, то и выделять ресурсы на расшифровку никто не станет.

Математических формул для точного подсчета стойкости не существует, предлагаются лишь те или иные оценки. Разумеется, стойкость зависит от длины ключа, чем ключ длиннее, тем большей полагается стойкость. Конечно, вычислительные возможности современных компьютеров растут очень быстро. Так, стандарт шифрования DES с 56-битовым ключом, разработанный в 1977 г. и применявшийся в государственных организациях США, к концу XX в. стал легкой добычей любителей, энтузиастов криптографии, объединивших для взлома сотни компьютеров по всему миру, работавших параллельно через Internet. Затраченное время измерялось "всего лишь" десятками часов.

Но и криптография не стоит на месте. Современные алгоритмы предлагают для перебора число вариантов, превышающее число атомов во Вселенной. Вероятность наткнуться на ответ исчезающе мала, поэтому такие алгоритмы называют *вычислительно стойкими*.

Аутентификация

Для информации, циркулирующей в защищенной системе, помимо секретности жизненно важна и другая задача — *аутентификации*. Аутентификация — способ доказать, что объект (человек, компьютерная система) — тот, за кого он себя выдает. Источник секретного сообщения должен быть установлен и проверен, иначе защита от навязывания ложных сообщений невозможна.

Применяют различные методы аутентификации:

- предъявление пароля;
- проверку неотъемлемых признаков, присущих опознаваемому объекту;
- доказательство обладания некоторой секретной (например, ключевой) информацией;
- установление факта присутствия объекта в заданном месте в заданное время;
- заключение третьей, доверенной, стороны.

Классические методы аутентификации с вековой историей — например, обычный бумажный паспорт или другой документ — уступают место новациям цифровой эры. Широкое распространение получают пластиковые смарт-карты (smart cards), содержащие подробную информацию о владельце,

а также считыватели отпечатков пальцев, устройства распознавания радужной оболочки глаз и другие биометрические приборы.

Вы проходите аутентификацию каждый день, регистрируясь в компьютерной сети или на локальном компьютере. За кажущейся простотой ввода пароля в Windows стоит достаточно сложный алгоритм, в котором пароль как таковой в целях безопасности вообще не пересылается по сети.

С аутентификацией тесно связана проблема *невозможности отказа от ответственности* (non-repudiation). В эпоху симметричных методов криптографии стороны, находящиеся на обоих концах засекреченного канала связи, использовали один ключ и полагались на взаимное доверие. Асимметричные методы позволяют расширить применение на тот случай, когда стороны могут не доверять друг другу. С их помощью можно не просто установить, но и неопровержимо доказать, что пересланное сообщение было подписано именно этим отправителем, и только им.

Целостность

Любое сообщение, посылаемое по Internet, может частично или полностью оказаться в чужих руках и подвергнуться корректировке. *Целостность данных* означает, что сообщение дошло до получателя в том виде, в каком оно было послано отправителем, без всяких (случайных или намеренных) изменений.

Для доказательства целостности в обычной переписке применяют запечатанные конверты. Аналогично поступают с цифровыми данными. К собственно передаваемому сообщению добавляют специальный контрольный код, или иначе хэш (hash), или дайджест. *Хэш* — это последовательность данных небольшой (по сравнению с сообщением) фиксированной длины, играющая роль печати.

Примечание

В современной криптографии применяются хэши длиной 128, 160 и 192 бита.

Хэш-функция, генерирующая такой код, должна обладать двумя свойствами.

- Она должна быть вычислительно необратимой. Это означает, что для любого сообщения можно однозначно вычислить значение его хэша, но, имея хэш, восстановить по нему исходное сообщение нельзя (за исключением, разумеется, полного перебора вариантов).
- Вероятность совпадения значений хэша для разных сообщений должна быть очень мала.

Кроме того, оппонент не должен иметь возможности подсчитать правильное значение хэша для своего ложного сообщения и тем самым выдать его за

подлинное. Поскольку ранее идея секретных алгоритмов нами была фактически отвергнута, следует сделать простой вывод: алгоритм вычисления хэша должен зависеть от переменного параметра — ключа. Только должные отправитель и получатель — владельцы ключа — могут вычислить хэш от данного сообщения и подтвердить его целостность.

Цифровые подписи, сертификаты и их применение

Революция в сфере аутентификации и проверки целостности произошла с приходом асимметричных криптографических методов. Вычислять и применять хэш математики научились очень давно, изюминка в том, чтобы использовать *подписанный хэш*.

Ранее мы говорили, что схема с двумя ключами может выполнять шифрование как открытым, так и секретным ключом. В первом случае зашифровать может каждый, прочитать — лишь один. Во втором случае, наоборот, зашифровать сообщение может только единственный обладатель секретного ключа. Зато расшифровать могут все обладатели открытого ключа. Термин "шифрование" здесь неприменим, это, скорее, процесс подписи. Таким образом, одни и те же алгоритмы могут применяться как для обеспечения секретности, так и для получения *цифровой подписи*.

Сообщение, подписанное секретным ключом, однозначно указывает на человека, который его изготовил (подписал), — точно так же, как и обычная подпись на документе.

Примечание

Некоторые указывают на существенную разницу между обычной и цифровой подписью. Обычная подпись всегда и при любых обстоятельствах принадлежит одному и тому же индивидууму. Цифровая подпись — это всего лишь данные, хранящиеся на компьютере, и владельцем ее является тот, кто в настоящий момент владеет этим компьютером. Вот почему иногда предлагают именовать это "цифровой печатью", которая может быть передана, украдена и т. п.

Для устранения этой коллизии в законодательстве некоторых стран разрешается иметь цифровую подпись только физическим лицам, но не юридическим. В других странах допускаются оба варианта.

Основные этапы создания подписанного документа показаны на рис. 36.2.

1. Отправитель выбирает алгоритм хэширования и формирует хэш документа.
2. Определяется ключевая пара и алгоритм шифрования, которые будут использоваться для подписи.

3. Полученный хэш обрабатывается (подписывается) секретным ключом отправителя.
4. В отправляемое сообщение включаются: тело документа, подписанный хэш, идентификаторы хэш-функции, функции кодирования и открытого ключа отправителя.

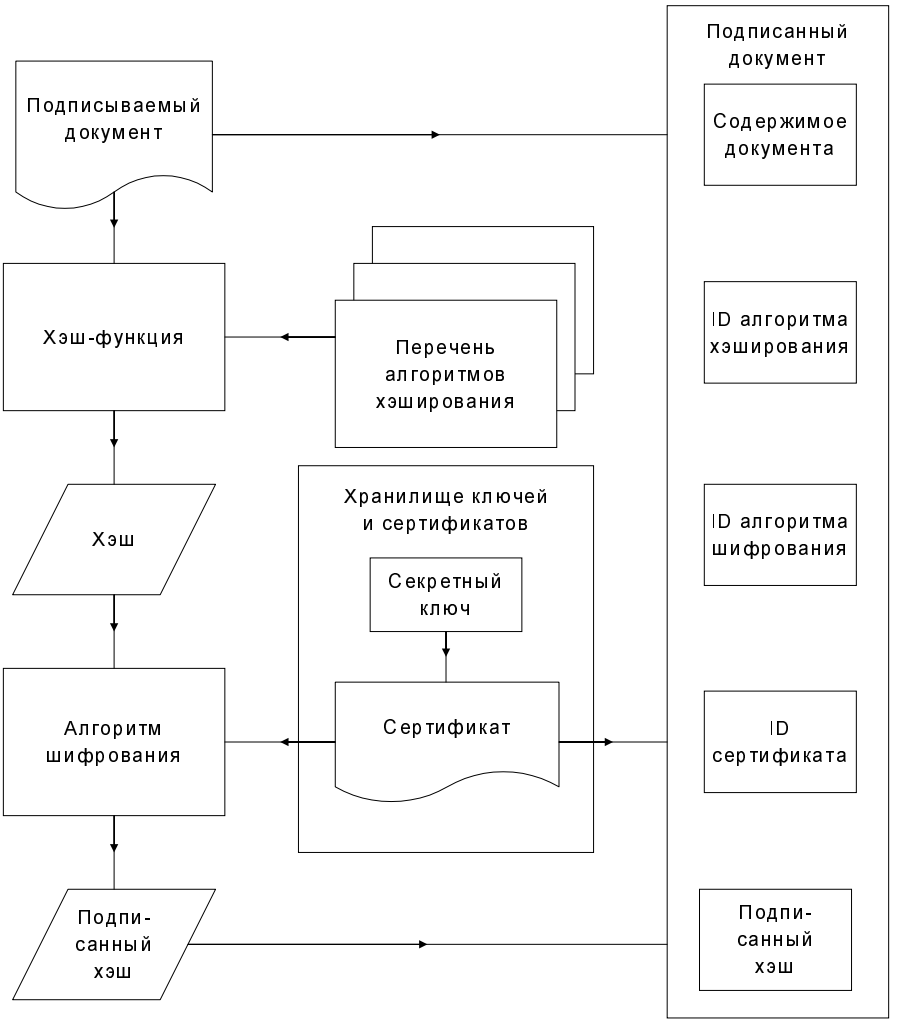


Рис. 36.2. Процедура получения цифровой подписи

На приемной стороне сначала происходит вычисление хэша по тому же самому алгоритму, что и у отправителя. Затем, зная открытый ключ отправи-

теля, мы расшифровываем значение хэша, сгенерированное отправителем до передачи. Если эти два значения совпадают, это означает, что сообщение доставлено без искажений и изъятий.

В предложенной схеме лишь один маленький изъян — аутентификация автора подписи. Подпись проверяется открытым ключом. Но где гарантии, что этот полученный вами открытый ключ действительно принадлежит автору подписи?

На ранних этапах становления цифровой подписи для публикации открытого ключа использовались разные методы максимально широкого и публичного охвата — вплоть до того, что ключ в двоичном виде публиковался на обложках книг, прикреплялся к каждому из тысяч отправляемых писем электронной почты и т. п. Это, разумеется, затрудняло потенциальным злоумышленникам подмену ключа.

Но полностью проблема была решена с появлением цифровых сертификатов (далее в этой книге — просто сертификатов). *Сертификат* представляет собой информацию о человеке или организации, плюс его открытый ключ, подписанную доверенной стороной — центром сертификации (ЦС, Certification Authority, CA). ЦС выдает сертификат, проведя аутентификацию, т. е. предварительно убедившись в том, что заявитель подал о себе верную информацию и действительно является владельцем данной ключевой пары.

Но почему мы должны доверять ЦС, и кто проводит их аутентификацию? Хороший вопрос, ответ на который существует лишь частично. Центры сертификации упорядочены и имеют свою иерархию. Каждый ЦС имеет свой сертификат, который должен быть выдан и подписан вышестоящим ЦС. Но кто находится на самом верху иерархии? Это относительно небольшая группа компаний с устоявшейся репутацией, делающих бизнес на проверке и выдаче сертификатов. Номер один среди них — компания VeriSign (**www.verisign.com**).

Поскольку репутация стоит денег, выдача сертификатов, как правило, процесс не бесплатный. Совсем мало или ничего не придется заплатить частному лицу за персональный сертификат. Более того, в ряде европейских стран, например, в Швеции, государство само разворачивает программы по выдаче сертификатов своим гражданам.

Другое дело — выдача сертификатов компаниям для коммерческих целей, и особенно дороги сертификаты для дочерних Центров сертификации.

О масштабах бизнеса в области сертификации говорит, например, тот факт, что основатель известного ЦС Thawte Inc. (ЮАР) Марк Шаттлворт в апреле 2002 г. стал вторым космическим туристом, заплатив несколько десятков миллионов долларов за недельный визит на Международную космическую станцию.

В этом месте совершим небольшой практический экскурс. Откройте ваш IE (Internet Explorer), и перейдите на вкладку **Content** (Содержание) диалогового окна **Internet Options** (Свойства обозревателя) (пункт меню **Tools | Internet Options** (Сервис | Свойства обозревателя)). На ней есть кнопка **Certificates** (Сертификаты), открывающая менеджер сертификатов.

Примечание

Этот же менеджер в виде отдельной программы (CertMgr.exe) поставляется в составе Microsoft Platform SDK.

Главная часть окна менеджера занята блокнотом с четырьмя страницами: **Personal** (Личные), **Other People** (Другие пользователи), **Intermediate Certification Authorities** (Доверенные издатели), **Trusted Root Certification Authorities** (Доверенные корневые центры сертификации). Они соответствуют четырем предустановленным в системе хранилищам сертификатов. На каждой странице — список сертификатов, находящихся в данном хранилище (рис. 36.3).

Страница **Trusted Root Certification Authorities** (Доверенные корневые центры сертификации) отведена для сертификатов корневых ЦС (самого верхнего уровня), которые включены в состав дистрибутива системы изначально и автоматически регистрируются. Добавлять и удалять сертификаты из этого списка может только пользователь Windows, имеющий административные полномочия. В число корневых входит и сертификат Microsoft.

Примечание

Фирма Microsoft не предоставляет коммерческие услуги по сертификации в обычном понимании, т. е. для аутентификации людей и организаций. Однако поставщики решений для Windows (драйверов аппаратуры, криптографических систем и т. п.) обязаны получить сертификат от Microsoft. В ОС Windows 2000 цифровая подпись драйвера проверяется, но ее наличие необязательно. В Windows XP все драйверы должны быть подписаны.

Сверху от блокнота расположен список выбора **Intended purpose** (Назначение), позволяющий фильтровать множество сертификатов. Что же такое назначение сертификата? Это хранящийся в нем перечень задач, для которых он может быть использован:

- аутентификация сервера;
- аутентификация клиента;
- подписывание исполняемого кода;
- защищенная электронная почта;
- удостоверение меток времени;
- криптографические средства Windows 2000: IPSec, EFS;
- подписывание драйверов и других составных частей Windows.

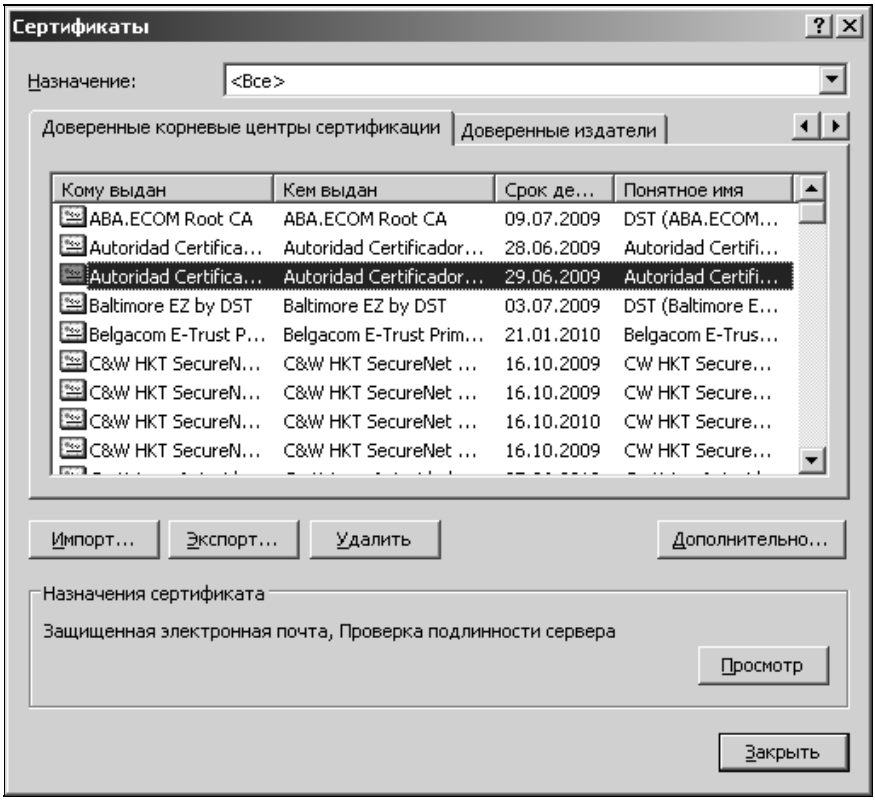


Рис. 36.3. Окно просмотра списка сертификатов в корневом хранилище

Полный список возможных задач можно увидеть, нажав кнопку **Advanced**.

Безопасность — это всегда отсечение лишних возможностей, поэтому использование сертификатов, как правило, ограничено одной или несколькими родственными задачами.

Дважды щелкнув на строке в таблице, можно открыть специальное окно для просмотра свойств сертификата и "покопаться" в нем (рис. 36.4).

Из трех страниц открывшегося диалогового окна первая представляет общую информацию в доступном даже для новичка виде, третья — всю цепочку сертификатов вплоть до корневого. Сосредоточимся на второй странице. Приведенный на ней список полей — это и есть все содержимое сертификата.

Немного о том, какими стандартами регулируется сфера сертифицирования. Стандарт формата цифровых сертификатов утвержден рекомендацией ITU-T (International Telecommunications Union) X.509, те же требования определены в документе ISO/IEC 9594-8. Этот стандарт действительно стал общепринятым.

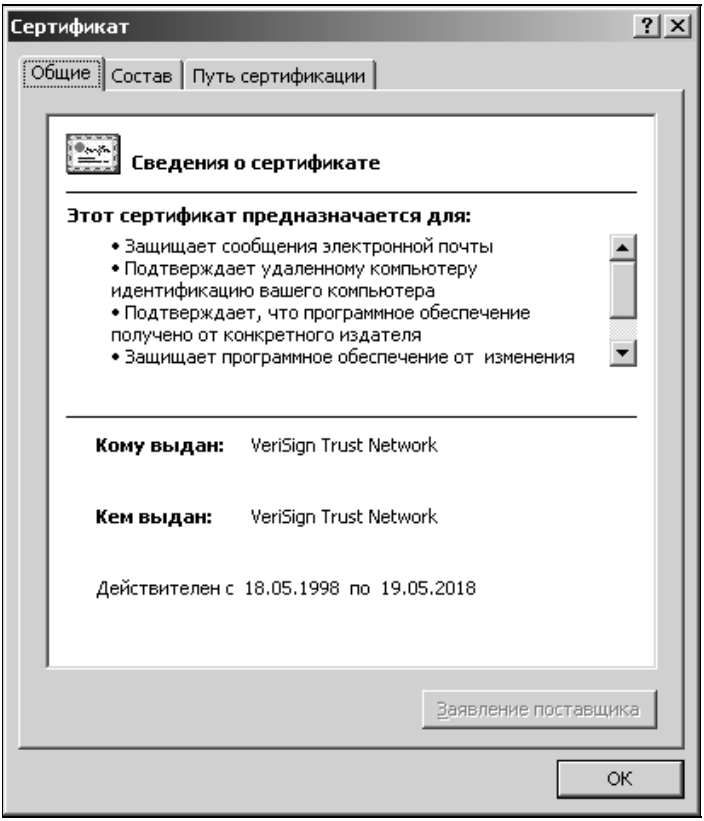


Рис. 36.4. Страница детального просмотра свойств сертификата

Кроме того, большой вклад в формирование стандартов внесла фирма RSA (по начальным буквам фамилий Rivest — Shamir — Adelman, создатели и владельцы патента на один из первых методов криптографии с открытым ключом). Сейчас срок патента истек и методы RSA опубликованы (см. **www.rsa.com**). Из разработанных ими стандартов PKCS (Public Key Cryptography Standard) вы, скорее всего, встретитесь с PKCS #10 (стандарт запроса на сертификат, посылаемый клиентом центру сертификации) и PKCS #7 (ответ на этот запрос, содержащий выданный сертификат). Дело в том, что эти стандарты поддерживаются Службой сертификатов Microsoft.

Согласно X.509, каждый сертификат должен иметь набор обязательных полей (табл. 36.1); кроме этого, допускаются необязательные расширения (идентификаторы ЦС и владельца сертификата, флаги использования сертификата и т. п.).

Таблица 36.1. *Содержание полей сертификата стандарта X.509*

Название	Содержание
`Version`	Номер версии сертификата — V1..V3. Сертификационный сервер Microsoft, который будет рассматриваться в этой главе, выдает сертификаты V3
`Serial Number`	Индивидуальный номер, уникальный для сертификатов, выданных данным центром сертификации. Может иметь переменную длину
`Signature algorithm`	Использованный алгоритм цифровой подписи
`Issuer`	Имя центра сертификации, выдавшего сертификат
`Valid from`	Дата и время, начиная с которого сертификат действителен
`Valid to`	Дата и время, вплоть до которого сертификат действителен
`Subject`	Имя человека или организации, которым выдан данный сертификат. Это поле часто включает дополнительную информацию — данные о стране, регионе, подразделении
`Public key`	Открытый ключ — двоичные данные длиной 512 или 1024 бита

Информация в сертификате хранится в определенной кодировке, различной для разных версий и для разных полей. О том, как ее извлечь и использовать, мы поговорим в следующем разделе, когда будем рассматривать демонстрационное приложение.

Использование *CryptoAPI*

Программный интерфейс `CryptoAPI` был впервые включен в поставку Microsoft Internet Explorer 3.02. Формальный повод связать криптографию и популярный Web-браузер — обеспечение защищенных соединений между HTTP-клиентом и сервером. Неформально эта история относится к периоду тяжб с Netscape, когда фирма Microsoft вынуждена был доказывать суду, что Internet Explorer является неотъемлемой частью системы. Привязка к браузеру такой важной составляющей, как криптография, несомненно была сильным козырем.

До определенного времени в ОС от Microsoft `CryptoAPI` не применялся практически нигде, кроме браузера. Однако с выходом Windows 2000 ситуация кардинально изменилась. Теперь криптография служит для создания шифрующей файловой системы (Encrypting File System, EFS), в защищенном варианте транспортного протокола IPSec (Internet Protocol Security), для удаленной авторизации по протоколу Kerberos, в шифрованной электронной почте и т. п.

В этой главе мы постараемся рассмотреть два аспекта работы с CryptoAPI. Во-первых, это структура самого интерфейса и его применение в ваших программах на Delphi. Во-вторых, это настройка и использование готовых криптографических средств операционной системы, которые могут понадобиться вам в создании распределенных информационных систем.

Структура *CryptoAPI*

CryptoAPI представляет собой двухуровневую структуру. На верхнем уровне находятся две системные DLL: advapi32.dll и crypt32.dll, содержащие подавляющее большинство функций CryptoAPI. Именно к ним происходит обращение из программ пользователя. Однако эти библиотеки сами не кодируют и не подписывают информацию, они всего лишь переадресуют вызовы на другой уровень.

На нижнем уровне находится *поставщик криптографических услуг (Cryptographic Service Provider, CSP)*. Это — "черный ящик", к которому обращаются криптографические функции для того, чтобы реально зашифровать, подписать данные и т. п. Механизм же реализации криптографических алгоритмов в этом "черном ящике" скрыт от пользователя — это может быть программный, аппаратный или смешанный комплекс. Каждый поставщик представляет собой динамическую библиотеку или набор библиотек, поддерживающих API более низкого уровня — CryptoSPI.

Эту схему построения можно сравнить с современными 3D-акселераторами графики (такое сравнение любит приводить фирма Microsoft). Действительно, программируя с использованием API OpenGL или Direct3D, вы не задумываетесь, как именно выполняются те или иные методы трехмерной графики. Большинство из них исполняется аппаратно графическим процессором на видеокарте, таким как Riva GeForce или ATI Radeon. Но если ваша видеокарта не поддерживает всех возможностей аппаратно, это не означает, что они вообще недоступны, в этом случае они будут эмулироваться программно.

В стандартную поставку Windows 2000 включаются несколько CSP, причем не все из них являются полнофункциональными. Кроме набора поддерживаемых функций, могут варьироваться используемые алгоритмы подписи/шифрования, длина ключа и т. п. Так, у исторически первого и распространенного провайдера MS Base Cryptographic Provider реализовано 40-битное симметричное шифрование и шифрование с открытым ключом длиной 512 бит. На его основе созданы Strong и Enhanced Cryptographic Provider, у которых длина ключей достигает соответственно 128 и 1024 бит.

Разработать своего поставщика криптографических услуг теоретически может каждый. Однако для того, чтобы он был установлен и работал в Windows, вы должны предоставить Microsoft полную информацию, пройти сертификацию и получить от них цифровую подпись. Разумеется, все это

процедуры весьма сложные и для обычного пользователя практически нереальные, поэтому приемлемых вариантов работы со средствами криптографии всего два: либо использовать CryptoAPI с множеством существующих провайдеров, либо применять решения от независимых поставщиков.

Для программистов, пишущих на C и C++, в составе Platform SDK и других продуктов для разработчиков поставляется заголовочный файл WinCrypt.h. Для программистов на Visual Basic и других скриптовых языках предлагается средство CAPICOM — объект Автоматизации, представляющий собой COM-надстройку (wrapper) над CryptoAPI.

Одна из наиболее сложных и нетривиальных функций CryptoAPI — это хранение ключей. Именно это является узким местом многих криптографических систем, ведь если злоумышленник может без труда пробраться к секретным ключам и заполучить их, то все замечательные и сложные алгоритмы шифрования оказываются бесполезными. В каждой новой ОС фирмы Microsoft методы хранения ключей усложняются. В Windows 95 они хранились в системном реестре. В Windows 2000 ключи хранятся в виде файлов внутри профиля пользователя, но при этом они зашифрованы специальным сессионным ключом, который создается и поддерживается специальной службой Protected Storage на основе данных о пользователе и компьютере.

Разумеется, эта информация нужна не для конкретного использования в программе, а для лучшего понимания сущности работы CryptoAPI. Для каждого пользователя в системе создается ключевой контейнер — поименованная часть общего хранилища ключей. Это логическая сущность, позволяющая программисту абстрагироваться от того, где физически находятся ключи.

Примечание

Иногда в целях безопасности и надежности следует все же не доверять и CryptoAPI. Секретные ключи, которые необходимы редко (например, ключ аварийного восстановления шифрованной файловой системы EFS), рекомендуется переписать на съемный носитель информации в нескольких экземплярах, поместить их в надежные места и затем стереть ключи с компьютера; в особенности это касается переносных компьютеров, которые чаще попадают в руки злоумышленников.

Итак, программист может найти для себя в CryptoAPI следующие полезные возможности:

- работу с провайдерами (получение списка, выбор, открытие и создание контекста доступа к ключевому контейнеру);
- основные криптографические операции (шифрование, расшифровка, подпись, проверка подписи);

- управление ключами (генерация симметричных ключей и асимметричных ключевых пар, экспорт/импорт, удаление);
- управление сертификатами (работа с хранилищами, просмотр и поиск в хранилищах, экспорт/импорт);
- проверку подлинности сертификата (работа со списками доверяемых и отозванных сертификатов).

Общее количество функций в `CryptoAPI` достигает трех сотен, и с выходом новых версий и обновлений растет (текущая версия CryptoAPI — 2.0). Подробная документация имеется в главном информационном ресурсе Microsoft для разработчиков MSDN (**http://www.microsoft.com/rus/msdn/**); там нужные функции легко отыскать по префиксам crypt- и cert-. Все основные операции снабжены примерами на языке C.

Пример использования *CryptoAPI* – менеджер сертификатов

Для работы с `CryptoAPI` в Delphi необходим файл с описанием его функций. К сожалению, фирма Borland не включает в поставку Delphi трансляцию заголовочного файла WinCrypt.h. Поэтому придется полагаться на работу независимых разработчиков, выполнивших такую трансляцию самостоятельно. На сайте **http://codecentral.borland.com** можно найти два таких примера: ID 15979, by Massimo Maria Ghisalberti и ID 17597, by Oleg Starodub. Содержащиеся в них файлы соответственно wcrypt2.pas и wincrypt.pas дадут вам возможность обращаться к нужным функциям. Оба соответствуют `CryptoAPI` версии 2, хотя файл wcrypt2.pas и является неполным.

Наш пример похож на менеджер сертификатов, имеющийся в составе Windows и описанный ранее. Тем не менее мы сочли именно такой пример достойным рассмотрения, т. к. каковы бы ни были ваши действительные задачи, столкнуться с организацией и доступом к сертификатам все равно придется.

Пользовательский интерфейс менеджера предельно прост — компонент `TTabControl`, на котором расположен собственно список (компонент `TListView`), и несколько кнопок для выполнения основных функций (рис. 36.5).

Наш менеджер работает со стандартными системными хранилищами: **MY** (соответствует странице **Personal**, Личные менеджера сертификатов), **TRUST** (**Other People**, Другие пользователи), **ROOT** (**Trusted Root Certification Authorities**, Доверенные корневые центры сертификации), **CA** (**Intermediate Certification Authorities**, Промежуточные центры сертификации). Кроме того, начиная с Windows NT4 SP4, появилась функция `CertEnumSystemStore`, позволяющая получить список всех хранилищ на компьютере. В примере происходит проверка версии ОС, и если эта функ-

ция поддерживается, то число страниц блокнота `TTabControl`, соответствующих хранилищам, увеличивается до их фактического количества.

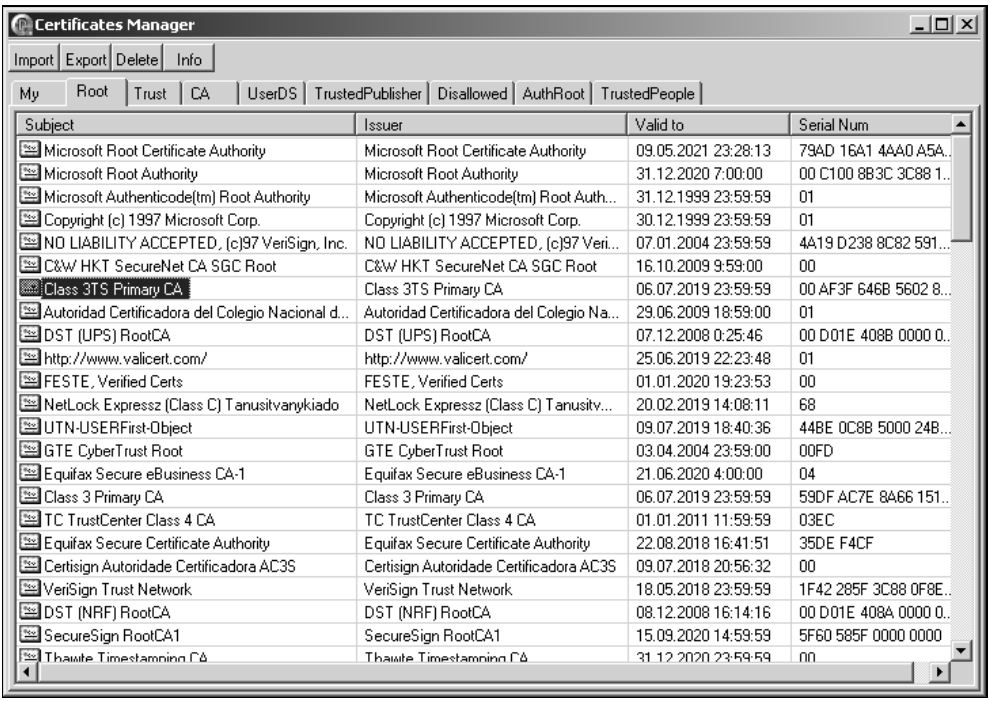


Рис. 36.5. Главное окно Менеджера сертификатов

Поля **Issuer** (Поставщик) и **Subject** (Субъект) кодируются в так называемом *стандарте DN* (Distinguished names, известные имена). Это строка, содержащая перечисленные через запятую атрибуты объекта, например: **'CN=MyServer, OU=UsersUnit'**. DN-поля, установленные для сертификатов, приведены в табл. 36.2.

Таблица 36.2. *Назначение DN-полей для сертификатов стандарта X.509*

Аббревиатура	Полное название	Содержание
CN	CommonName	Имя физического или название юридического лица; для сертификатов, выдаваемых компьютеру, — имя DNS
O	Organization	Юридическое название организации

Таблица 36.2 *(окончание)*

Аббревиатура	Полное название	Содержание
OU	Organizational Unit	Название подразделения, например, Certification Services Division
L	Locality	Город, где расположена организация
S	State or Province	Полное название штата или региона
C	Country	Двухбуквенная аббревиатура названия страны (US, CA, RU и т. п.)
E	E-mail	Адрес электронной почты

Два поля даты — **Valid From** и **Valid To** — кодируются в обычном для Windows формате представления времени.

Наконец, некоторые поля (идентификаторы алгоритмов) представлены так называемым Идентификатором объекта (Object Identifier, OID). Такая кодировка предусмотрена стандартом ASN.1 (Abstract Syntax Notation), применяемым в X.509. В этой кодировке любые константы представлены строкой с последовательностью целых чисел, разделенных точками; например, для алгоритма формирования хэша SHA1 используется строка '1.2.840.113549.1.1.5'. Но как получить из этой неудобочитаемой последовательности цифр приемлемое название? В `CryptoAPI` для этого существует функция `CryptFindOIDInfo`, которая по заданному OID выдает название закодированной величины.

Наконец, номер сертификата и открытый ключ представлены массивами байтов. Номер пригодится нам для поиска нужного сертификата в хранилище. Дело в том, что функция `CertFindCertificate` может искать сразу по многим признакам: по имени поставщика и субъекта, по примененным алгоритмам, по ключу и т. п. Но если вспомнить о том, что номер каждого сертификата у данного поставщика является уникальным, то комбинация "идентификатор поставщика + номер сертификата" обеспечивает однозначность поиска. Такая структура называется `CERT_ID`. При составлении списка сертификатов мы создаем экземпляр такой структуры и связываем его с полем **Data** каждого элемента `TListItem` компонента `TListView`.

В нашем примере есть функция получения подробной информации о сертификате, которая декодирует основные поля и представляет их в удобном виде; познакомившись с ней, вы научитесь принципам работы с этими разнообразными (и непривычными для обычного пользователя Windows) кодировками.

Основной структурой для всех функций работы с сертификатами является CERT_CONTEXT:

```
CERT_CONTEXT = record
    dwCertEncodingType: DWORD;
    pbCertEncoded:      PBYTE;
    cbCertEncoded:      DWORD;
    pCertInfo:          PCERT_INFO;
    hCertStore:         HCERTSTORE;
  end;
```

Как видно, структура CERT_CONTEXT имеет две основные составные части: собственно тело сертификата в закодированном виде (массив pbCertEncoded длиной cbCertEncoded байтов) и запись pCertInfo, которая включает в себя все основные поля и поля расширения.

В листинге 36.1 содержится описание только одной из задач менеджера сертификатов — импорта (полный листинг занял бы слишком много места). Весь исходный код примера имеется на прилагаемом к книге диске. Параметрами процедуры являются Store — имя системного хранилища, в которое мы пытаемся поместить сертификат, и FileName — имя файла с сертификатом. Файл должен иметь формат PKCS#7. Такие файлы, например, выдаются Службой сертификатов Microsoft .

Примечание

Для экспорта/импорта сертификата вместе с личным (секретным) ключом существует другой формат — PFX. Секретный ключ имеют только сертификаты, выданные ЦС лично вам (или вашему компьютеру).

Листинг 36.1. Менеджер сертификатов (процедура импорта)

```
function InstallCert(const Store, FileName: string): Integer;
var CertFile: file;
    hStore : HCERTSTORE;
    PKCS7Length: Integer;
    Buf:pointer;
    pccExisting, pccNewContext :PCCERT_CONTEXT;
    le : Integer;
    Res : boolean;
begin
 Result := -1;
 try
```

```
try
 AssignFile(CertFile, FileName);
 ReSet(CertFile, 1);
 PKCS7Length := FileSize(CertFile);
 buf := AllocMem(PKCS7Length);
 if not Assigned(buf) then RaiseLastOSError;
 {$I-}
 Blockread(CertFile, buf^, PKCS7Length, le);
 if PKCS7Length<>le then RaiseLastOSError;
 hStore := CertOpenSystemStore(0, pChar(Store));
 if not Assigned(hStore) then RaiseLastOSError;
 pccNewContext := CertCreateCertificateContext(X509_ASN_ENCODING or
 PKCS_7_ASN_ENCODING,
                                    buf, PKCS7Length);
 if not Assigned(pccNewContext) then RaiseLastOSError;
 pccExisting := CertFindCertificateInStore(hStore, X509_ASN_ENCODING or
 PKCS_7_ASN_ENCODING, 0,
 CERT_FIND_EXISTING, pccNewContext, nil);
 if Assigned(pccExisting) then
  begin
   CertFreeCertificateContext(pccExisting);
   Result := 0; //already exists
   Exit;
  end;
 Res := CertAddCertificateContextToStore(hStore,
                                    pccNewContext,
                                    CERT_STORE_ADD_NEW,
                                    nil);
 le := GetLastError;
 if (Res = False) and (DWORD(le)=$80092005)(*CRYPT_E_EXISTS*) then
  Result := 0
 else
  raise EOSError.CreateResFmt(@SOSError, [le, SysErrorMessage(le) ])
finally
 if Assigned(Buf) then FreeMem(Buf);
 CloseFile(CertFile);
 CertFreeCertificateContext(pccNewContext);
 CertCloseStore(hStore, 0);
end;
```

```
  except
   on E:EOSError do
    Result := E.ErrorCode;
   on E:EInOutError do
    Result := E.ErrorCode;
  end;
end;
```

В нашем примере пользовательский интерфейс и функции работы с сертификатами разделены и находятся соответственно в модулях Main и CertUtils. Надеемся, описанные утилиты смогут пригодиться вам и для других проектов.

Служба сертификатов Microsoft

Для создания системы, реализующей инфраструктуру криптографии с открытым ключом, не обязательно запасаться множеством сертификатов, выданных сторонними организациями. ОС Windows 2000 сервер включает в себя службу, которая поддерживает полный цикл работы с сертификатами: выдачу, поддержку и публикацию списков выданных и отозванных сертификатов и т. п.

Важно, что эта служба не зависит от транспортного протокола. Разработчику предоставляются COM-интерфейсы основных модулей службы сертификатов, которые можно использовать из различных клиентских приложений. Если у вас на сервере установлена такая служба, из вашего браузера обратитесь по адресу **http://SERVERNAME/CertSrv** (Web-папка с таким именем создается по умолчанию, возможно, ваш администратор присвоил ей другое имя). На рис. 36.6 приведен пример клиентского приложения службы сертификатов, которое использует в качестве интерфейса браузер.

Задачи, решаемые приложением, вполне типичны для взаимодействия с центром сертификации:

- получение и установка корневого сертификата ЦС;
- получение списка отозванных сертификатов;
- запрос на выдачу нового сертификата;
- проверка статуса запроса, посланного ранее.

Дело в том, что ЦС Microsoft может работать в двух режимах: немедленной и отложенной выдачи сертификатов. В первом случае запрос на сертификат обрабатывается сразу по получении, и, если он удовлетворяет всем услови-

ям, новый сертификат выдается в качестве выходного параметра соответствующего метода.

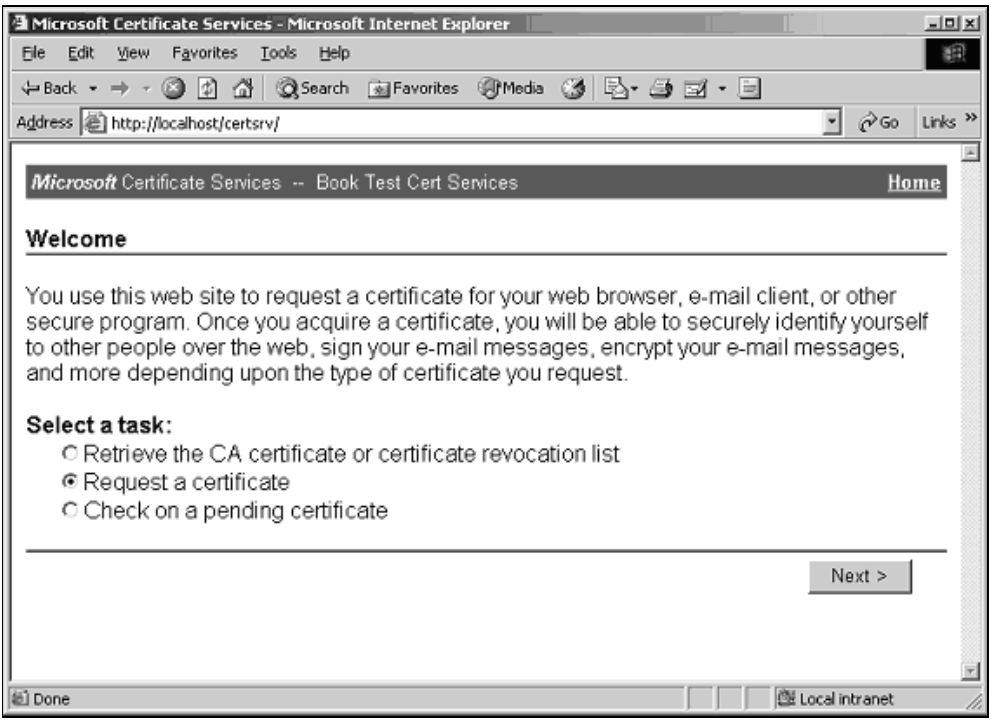


Рис. 36.6. Вид Web-интерфейса к службе сертификатов Microsoft

Во втором случае запросы ставятся в очередь, которая подлежит просмотру уполномоченным администратором с помощью специальной оснастки Certification services. Администратор лично разрешает выдачу сертификатов, отбраковывая нелегитимные запросы.

Примечание

Если ваш ЦС настроен на автоматическую выдачу и упомянутая Web-папка не имеет дополнительных средств защиты (скажем, авторизации по паролю), то получить сертификат от вашего ЦС может кто угодно. Это всегда как минимум нежелательно, а иногда и критично.

На серверной стороне за обработку запросов отвечает COM-объект `CertCli`, а на клиентской — `xenroll`.

Протоколы Internet для защищенных соединений

Одним из первых (1994 г.) был разработан протокол сетевой аутентификации SSL (Secure Sockets Layer). В 1996 г. обновленная версия протокола (3.0) была рекомендована для утверждения, поэтому не удивляйтесь, если встретите на официальном сайте SSL (**http://www.netscape.com/eng/ssl3**) именно эту версию многолетней давности. Но что же произошло с протоколом за последующие 6 лет? В 1999 г. он был утвержден в качестве официальной рекомендации (RFC 2246), но под названием TLS 1.0 (Tansport Layer Security) с минимальными исправлениями и добавлениями. Фирма Microsoft предпочитает в своих документах ссылаться на термин TLS, но большинство специалистов по-прежнему употребляют аббревиатуру SSL. Вы также можете встретить термин SChannel — это набор средств для Windows, обеспечивающий поддержку обоих протоколов-близнецов.

Давайте правильно определим место SSL в семиуровневом стеке протоколов OSI/ISO. Это надстройка над транспортным уровнем, которая отвечает за шифрование и подписание пакетов данных, генерацию и обмен ключами. Разумеется, для нормальной работы эти пакеты должны быть доставлены протоколом сетевого уровня (обычно TCP/IP). За использование шифруемой информации отвечает протокол более высокого уровня. Часто путают SSL и протокол HTTPS, но это не одно и то же: HTTPS — это вариант протокола уровня приложений HTTP, использующий для установки соединения SSL. Если для обычного HTTP выделен порт 80, то в HTTPS по умолчанию — порт 443.

Протокол SSL поддерживает два режима аутентификации: аутентификацию сервера и взаимную аутентификацию. В первом случае только сервер доказывает свою подлинность (с помощью своего сертификата), во втором такое же доказательство требуется от каждого клиента, устанавливающего соединение с этим сервером.

Как же происходит взаимодействие по протоколу SSL? При установке соединения происходит "рукопожатие", состоящее из четырех этапов; два выполняются клиентом, два — сервером.

1. Клиент посылает серверу запрос (ClientHello message), содержащий:
   - список криптографических алгоритмов для шифрования и подписи, поддерживаемых клиентом;
   - идентификатор сеанса, если клиент хочет восстановить ранее прерванный сеанс;
   - случайное число, используемое для инициализации начального состояния;
   - список алгоритмов сжатия, если такие будут применяться.

2. Сервер выбирает один из алгоритмов, чей перечень прислан клиентом (список упорядочен таким образом, что предпочтительные алгоритмы стоят в нем первыми). Если сервер не поддерживает ни один из них, посылается сигнал отказа от установления соединения. Также сервер посылает свой сертификат — основу для всех последующих действий, и еще одно случайное число.

   Если для соединения требуется взаимная аутентификация, на этом этапе сервер должен запросить у клиента его сертификат.

3. На основе упомянутых случайных чисел клиент генерирует сессионный ключ — секретный ключ для симметричного шифрования. Разрядность сессионного ключа может составлять 40 или 128 бит, в зависимости от настроек сервера. Сгенерированный ключ клиент подписывает открытым ключом сервера (который был получен в теле сертификата) и шифрует алгоритмом, который был предложен на этапе 1 и утвержден на этапе 2. Все это посылается серверу вместе со специальным признаком перехода в защищенный режим.

   Если сервер затребовал сертификат для взаимной аутентификации, клиент посылает его.

4. Сервер расшифровывает сессионный ключ своим секретным ключом. Если клиент прислал сертификат, происходит его проверка.

Все готово для установления двухстороннего секретного обмена данными. Сервер отправляет признак готовности, и в дальнейшем обе стороны посылают данные, зашифрованные сессионным ключом.

В этот момент, при установленной опции Internet Explorer **Warn if changing between secure and not secure mode** (Предупреждать о переключении режимов безопасности), пользователь видит следующее сообщение (рис. 36.7).

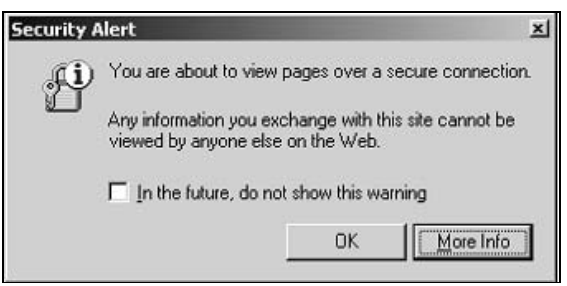


Рис. 36.7. Сообщение Internet Explorer, появляющееся на экране при переключении в режим SSL

Если пользователь IE хочет просмотреть информацию о сертификате сервера, который обеспечивал соединение по SSL, он может дважды щелкнуть на

значке, появляющемся в полосе состояния IE при работе в защищенном режиме (рис. 36.8).

Рис. 36.8. Значок в полосе состояния Internet Explorer, который появляется в режиме SSL и позволяет просмотреть сертификат сервера

Среди параметров безопасности IE, касающихся SSL, есть еще один — малозаметный, но очень важный. Речь идет о флажке с заголовком **Check for server certificate revocation (requires restart)** (Проверять аннулирование сертификатов серверов). Выбранный флажок заставляет клиента SSL перед установкой соединения запросить у центра выдачи сертификатов свежую версию списка отозванных сертификатов (Certificate Revocation List, CRL). Как любая чувствительная информация, например, те же пароли, — секретный ключ сертификата может быть украден. В этом случае недействительным становится и весь сертификат, ведь теперь злоумышленник может выдавать себя за истинного владельца. Единственное, что может в этом случае сделать потерпевший, — это известить ЦС об утрате. Все сертификаты, потерявшие легитимность до истечения срока их годности, отзываются и помещаются в CRL, предоставляемый ЦС по требованию всем желающим.

По умолчанию в IE флажок проверки списка CRL сброшен — ведь такие списки, по сути, являются базами данных и могут достигать очень больших размеров. Но если вы создаете защищенное приложение для работы во внутренней сети и передаете критически важную информацию, такая проверка может оказаться очень важной.

Примечание

Кстати, небольшой, но разумный срок годности сертификата — тоже отличное средство уменьшить возможные потери от утраты его легитимности.

Настройка SSL на стороне сервера IIS 6

Рассмотрим настройки защищенных Web-соединений на примере сервера Internet Information Services v.6.0, входящего в состав Windows Server 2003.

Как следует из описания принципов работы протокола SSL, приведенных ранее, сертификат сервера является абсолютно необходимым средством для

работы. Поэтому первое, что необходимо сделать для того, чтобы ваш сервер поддерживал защищенные соединения, — это получить для него сертификат от соответствующего центра сертификации. Вы можете сами получить этот сертификат от своего локального ЦС. Но этот вариант имеет существенный изъян: такой сертификат должен войти в список доверяемых на машинах всех клиентов. Иначе, в зависимости от настроек безопасности Internet Explorer, в худшем случае соединение с сервером будет отвергнуто, в лучшем — пользователю будет отправлен запрос на разрешение соединения (рис. 36.9). На него большинство пользователей ответит "нет" и поступит правильно.

Рис. 36.9. Окно соединения по HTTPS с сервером, чей сертификат не является доверяемым

Если вы делаете собственное распределенное приложение для локальной сети своей фирмы, то установить везде необходимые сертификаты не составляет особого труда. Однако если вы не знаете всех своих потенциальных клиентов, правильным решением будет потратить определенную сумму денег и получить сертификат сервера от одного из корневых ЦС, включенных по умолчанию в список сертификатов IE.

Отсутствие серверного сертификата можно сразу заметить по двум признакам:

- поле редактирования порта SSL недоступно (рис. 36.10);
- в группе элементов управления **Secure Communications** (Безопасные подключения) на странице **Directory Security** (Безопасность каталога) актив-

на только кнопка **Server Certificate** (Сертификат), открывающая доступ к мастеру установки сертификата (рис. 36.11).

Не подменяя этой главой документацию по IIS, кратко опишем шаги, необходимые для настройки SSL на сервере.

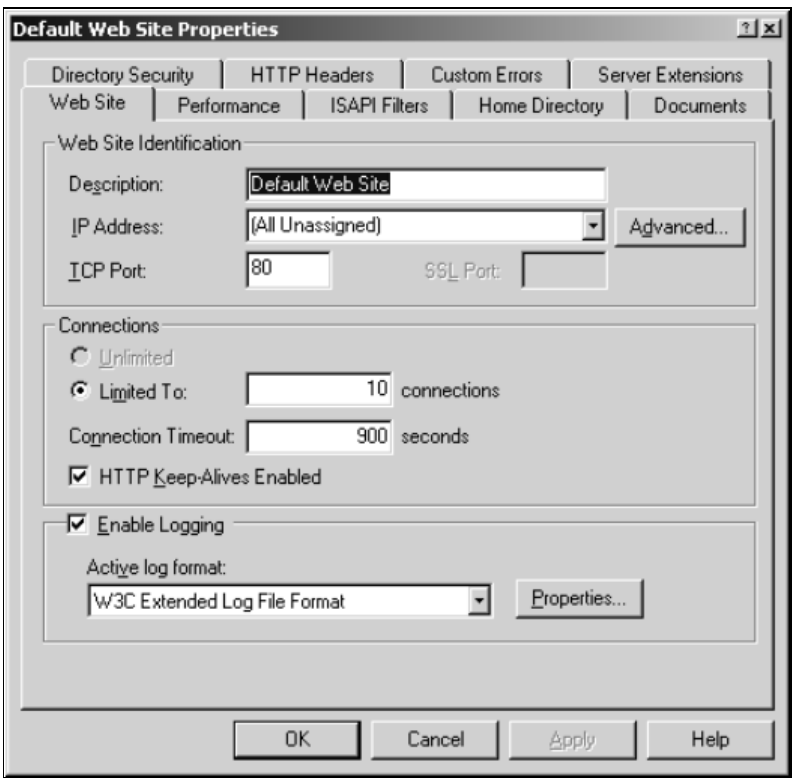


Рис. 36.10. Страница **Веб-узел** свойств Web-сайта, позволяющая настроить защищенное соединение

Сертификат сервера можно выбрать из уже установленных на компьютере подходящих сертификатов (т. е. содержащих в поле назначения флаг **Server Authentication** (Назначение существующего сертификата)).

Если вы хотите получить новый сертификат, то запустите специальный мастер создания нового сертификата. На самом деле этот мастер — просто средство для создания запроса в формате PKCS #10 с заранее прописанным назначением (аутентификация сервера). Далее этот запрос направляется в выбранный вами ЦС, и после получения ответа вы можете снова вызвать этот мастер, установить сертификат и связать его с Web-сервером.

При работе с мастером важно иметь в виду, что в поле "имя сайта" (**Your Site's Common Name** (Полное имя)) следует ОБЯЗАТЕЛЬНО задать тот же URL, который будет использоваться HTTP-клиентами для доступа по защищенному соединению. В случае если вы собираетесь сделать сервер открытым для доступа через Internet, нужно указать доменное имя сервера (**www.myserver.com**). Для внутренних сетей можно указать имя компьютера-сервера, в обоих случаях допустим также IP-адрес.

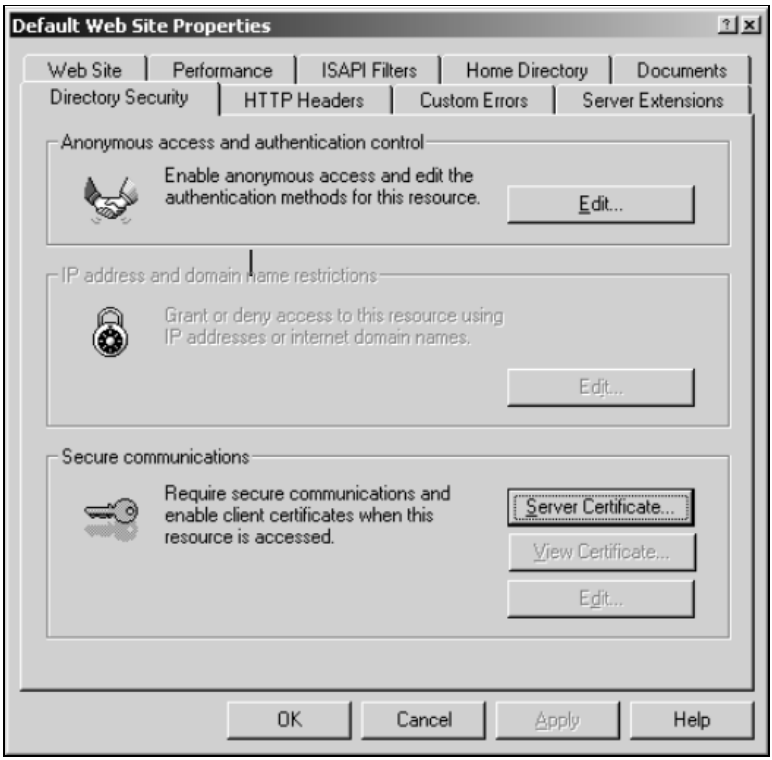


Рис. 36.11. Страница **Безопасность каталога** свойств Web-сайта, позволяющая настроить защищенное соединение

Есть и третий способ получения сертификата — импорт из файла *.key. В этом формате экспортировались сертификаты и соответствующие им секретные ключи в IIS версии 4 и более ранних.

Страница **Directory Security** (Безопасность каталога) и, соответственно, настройки SSL имеются в свойствах каждой папки Web-сайта IIS, но кнопка **Server Certificate** (Сертификат) на них неактивна. Это означает, что в пределах всего сайта действителен один общий сертификат, но параметры SSL вы

можете настраивать для папок индивидуально. Если вы хотите закрывать часть информации при помощи другого сертификата, вам придется создать в IIS отдельный сайт и связать его с этим сертификатом.

Настройки SSL можно менять после нажатия кнопки **Edit** (Изменить) (рис. 36.12).

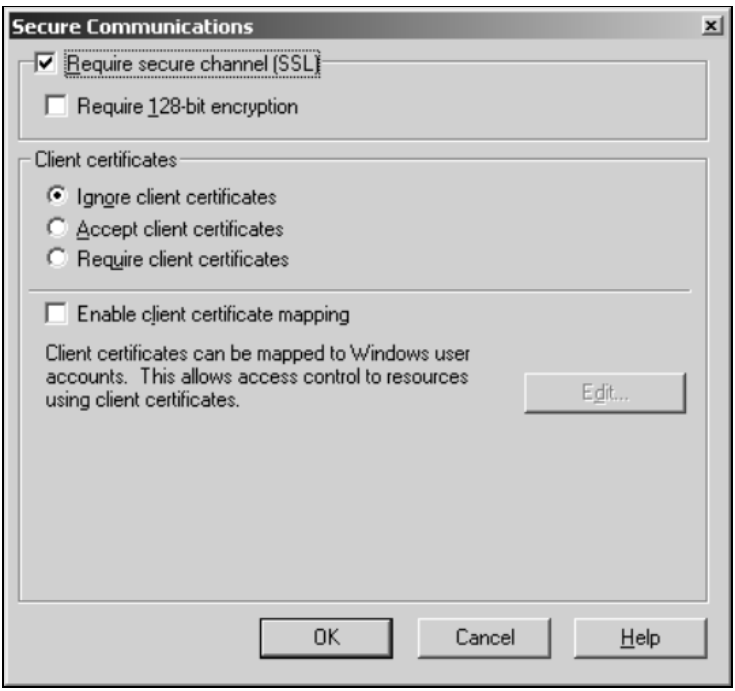


Рис. 36.12. Настройки SSL для Web-папки в Internet Information Services 6

Почти все настройки мы уже рассмотрели, за исключением единственного специфичного для Windows флажка **Enable client certificate mapping** (Разрешить сопоставление сертификатов клиентов), позволяющего отображать сертификаты, которыми подписались клиенты, на учетные записи пользователей Windows для упрощения управления ресурсами.

Настройка протокола SSL на клиентской стороне

Клиентская сторона в нашем рассказе будет представлена браузером фирмы Microsoft. Даже если ваше приложение-клиент не основано на браузере,

скорее всего, взаимосвязь с теми или иными составными частями есть. Например, реализация протокола SOAP для Delphi базируется на библиотеке WinInet, которая представляет собой ни что иное, как реализацию протоколов HTTP и FTP для Internet Explorer, и, разумеется, разделяет с ним все параметры и настройки.

Примечание

Конкурирующие браузеры также имеют свою реализацию HTTPS. Более того, изначально протокол SSL был разработан и реализован Netscape. Для хранения и управления сертификатами Netscape Navigator использует собственные функции и хранилища, никак не связанные с `CryptoAPI`.

Обычно когда требуется только аутентификация сервера, для установления соединения достаточно того, чтобы сертификат сервера был не просрочен, не отозван и выдан тем центром сертификации, которому клиент доверяет. Если это так, то клиенту надо только изменить в соответствующем URI названия протокола с **http://something/...** на **https://something/**. Клиенты, которые будут пытаться получить доступ к ресурсу по-старому, получат ошибку HTTP 403.4 (Forbidden: SSL required).

Кроме сертификата сервера, в защищенном обмене данными могут принимать участие и сертификаты другой стороны — клиентские, удостоверяющие личность клиента вашего приложения. Информация о сертификате передается в запросе и может быть прочитана сервером. Это удобно для организации систем, право доступа к которым имеет ограниченный круг пользователей. Предположим, вы запустили собственный ЦС и выдали сертификаты только тем пользователям, которые допущены к работе с вашим приложением. В этом случае вам надо выбрать радиокнопку **Require client certificates** (см. рис. 36.12). Доступ к вашему HTTP-ресурсу получат только обладатели сертификатов, прочие клиенты будут отвергнуты с кодом ошибки HTTP 403.7 (Forbidden: Client certificate required).

Если на вашем компьютере имеется более одного сертификата, по своему назначению пригодного для авторизации клиента, то браузер покажет диалоговое окно со списком пригодных для этого сертификатов (рис. 36.13).

Примечание

После получения сертификата online или изменения настроек часто бывает необходимо закрыть все экземпляры IE на компьютере.

Несколько слов для тех, кто собирается использовать SSL без технологий Microsoft.

Пакет OpenSSL поддерживает, как и средства Microsoft, протоколы SSL версий 2 и 3 и TLS 1.0. Он постоянно обновляется.

Примечание

Решения на основе SSL подпадают под правила ограничения экспорта в США (U.S. Export Restrictions). Перед тем как применять и распространять криптографические решения, обязательно ознакомьтесь с правовыми аспектами этой проблемы.

Рис. 36.13. Диалоговое окно запроса клиентского сертификата для соединения по SSL

Если вы устанавливаете связь по HTTPS не интерактивно из браузера, а через протокол SOAP, вам будет необходимо программно указать на клиентский сертификат. Как это делается, вы узнаете в следующих главах.

Резюме

В этой главе рассмотрены общие принципы построения защищенных информационных систем на основе современной криптографии. Тематика эта сравнительно нова. С одной стороны, издано много хороших книг с описанием научных теорий криптографии, с другой, есть руководства по безопасности для системных администраторов с простыми методиками

вида "делай раз — делай два". На наш взгляд, эта глава может восполнить пустоту между этими полюсами и научить не просто применять, но понимать криптографию.

Разумеется, требования безопасности не сводятся только к применению криптографии. Проблема эта многосторонняя, и вам придется отслеживать множество вопросов, которые выходят за рамки этой книги — от своевременной установки заплаток и пакетов обновления до ограничения физического доступа к компьютерам.

Вне поля зрения этой главы остался также защищенный протокол сетевого уровня IPSec (Internet Protocol Security). Этот относительно новый протокол впервые реализован Microsoft в Windows 2000. Помимо Windows он применяется и в ОС других производителей, но пока он поддерживается не так широко, как SSL. Если все клиенты ваших приложений поддерживают IPSec, вам придется искать подробные сведения о принципах его применения в документации Windows 2000 Resource Kit.

ЧАСТЬ X

Распределенные Web-приложения и Web-службы

Глава 37

Серверные Web-приложения

Феномен Internet оказал на развитие распределенных приложений настолько сильное влияние, что привел к появлению самостоятельного типа распределенных приложений, использующих технологии обмена данными Internet для собственных нужд. В таких приложениях применяется транспортный протокол HTTP, клиентские приложения с пользовательским интерфейсом на основе HTML обмениваются с Web-сервером запросами и ответами. Необходимым дополнением является специализированное приложение (аналог приложений промежуточного уровня), которое взаимодействует с Web-сервером, получая от него запросы, обрабатывая их и возвращая ответы для последующей передачи клиенту. Для таких приложений в документации Borland существует общее название — *серверные Web-приложения*. И в дальнейшем мы будем придерживаться этого термина.

Для разработки серверных Web-приложений в состав Delphi включены несколько технологий. Базовыми являются технологии WebBroker и WebSnap. Они реализуют полный цикл разработки и похожи не только названиями, но и по существу. Можно даже сказать, что технология WebSnap является развитием технологии WebBroker, т. к. включает все основные возможности второй и, плюс к этому, имеет некоторые дополнительные. К примеру, в WebSnap разработчик может легко встраивать в приложения скрипты на языках JavaScript и VBScript.

Специально для создания серверных Web-приложений, ориентированных на работу с базами данных, в Delphi имеется технология InternetExpress. Она объединила возможности технологий WebBroker и WebSnap по разработке серверной части Web-приложения и Web-интерфейса пользователя с эффективным способом обмена данными и богатыми возможностями обработки данных технологии *DataSnap (см. часть VII)*.

И, наконец, технология IntraWeb позволяет создавать пользовательский интерфейс Web-приложений не путем трудоемких манипуляций с тегами HTML, а привычным для разработчиков Delphi способом — визуальным программированием. В результате пользовательский интерфейс приложений

на основе IntraWeb получается более изящным, функциональным и приближающимся к стандартам Windows GUI.

Все перечисленные технологии программирования будут рассмотрены в главах этой части. В этой же главе мы обсудим следующие вопросы.

- Архитектура серверных Web-приложений.
- Взаимодействие серверного Web-приложения с Web-сервером.
- Доступные типы Web-серверов.
- Обработка запросов и формирование ответов.
- Динамическая генерация страниц HTML и XML.
- Отладочный сервер Web App Debugger.

Архитектура серверных Web-приложений

Основная отличительная черта распределенных Web-приложений заключается в том, что для выполнения большинства важнейших функций в них используются возможности стандартных инструментов Internet.

Во-первых, Internet является транспортной средой для таких приложений. Они применяют стандартные протоколы HTTP, HTTPS, SMTP и др. Местоположение серверов приложений определяется их IP-адресами.

Во-вторых, клиентские части таких приложений основываются на возможностях браузеров Internet. А пользовательский интерфейс клиентов строится на базе языков HTML и XML.

Таким образом, распределенное Web-приложение состоит из следующих частей.

- Клиентские части на основе Web-браузеров.
- Web-серверы.
- Серверные Web-приложения.
- Серверы БД.

Клиентские приложения при помощи специализированных Web-браузеров взаимодействуют с Web-серверами, доступными в Internet. Web-серверы занимаются приемом и диспетчеризацией запросов и передачей ответов клиентам. Однако запрос предназначен не Web-серверу, который является лишь частью механизма передачи запросов и ответов, а серверному Web-приложению. Для этого запрос должен содержать информацию, достаточную для его передачи запрашиваемому серверному Web-приложению.

Бóльшая часть Web-серверов работает под управлением UNIX и популярного на этой платформе сервера Apache. Фирма Microsoft также предостав-

ляет сервер Microsoft Internet Information Server на платформах Windows NT Server и Windows 2000 Server (в других ОС фирмы Microsoft есть его упрощенная версия).

Как видите, распределенные Web-приложения действительно во многом базируются на стандартных инструментах Internet (рис. 37.1). Предметом разработки является только серверное Web-приложение, которое по существу содержит всю бизнес-логику распределенного приложения. Оно принимает запрос и обрабатывает его, формируя ответ и закладывая в него запрошенную информацию. Для получения необходимой для создания ответа информации серверное Web-приложение должно уметь взаимодействовать с сервером БД. Поэтому в дальнейшем изложении мы не будем разделять понятия распределенного Web-приложения в целом и его части — серверного Web-приложения.

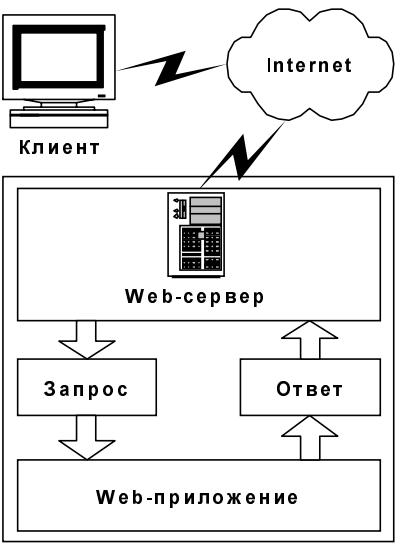


Рис. 37.1. Архитектура распределенного Web-приложения

Как это часто бывает, основное преимущество описанной архитектуры распределенного приложения выливается в его главный недостаток. Если вы обратитесь к главам предыдущей части, то увидите, что механизм передачи данных скрыт в недрах технологии DataSnap, а сами данные упаковываются на сервере, передаются в пакетах, распаковываются на стороне клиента и отображаются в стандартных визуальных компонентах стандартным образом. Здесь же серверное приложение должно отправлять ответы клиентам в формате HTML, пригодном для отображения средствами Web-браузера.

В этом и заключается главная проблема — при традиционных способах формирования ответов серверные Web-приложения должны хранить тысячи и тысячи страниц HTML. И страшно подумать, что будет, если, к примеру, понадобится изменить заголовки хотя бы части из этой массы страниц HTML.

Решение проблемы заключается в динамической генерации страниц ответов на основе информации, извлеченной из базы данных, и *шаблонов страниц*. Серверное Web-приложение должно уметь обработать запрос клиента, при необходимости обратиться к серверу БД за требуемой информацией и после получения результатов оформить ответ клиенту, вставив данные в шаблон (заготовку) страницы ответа.

Помимо тегов HTML страницы ответов могут включать *сценарии* или *скрипты*. Задача такого скрипта — реализация бизнес-правил на клиентской стороне и обеспечение взаимодействия с пользователем. Для разработки скриптов созданы и используются многие специализированные языки. В рамках технологий, реализованных в Delphi, можно применять скрипты, созданные на основе языков сценариев VBScript и JScript.

Типы серверных Web-приложений

В Delphi вы можете создавать серверные Web-приложения нескольких типов. Дадим им краткую характеристику.

Для создания серверного Web-приложения необходимо в Репозитории Delphi перейти в категорию **Delphi Projects | WebBroker** и выбрать значок **WebServer Application**. После этого появляется диалог, в котором необходимо выбрать тип приложения (рис. 37.2).

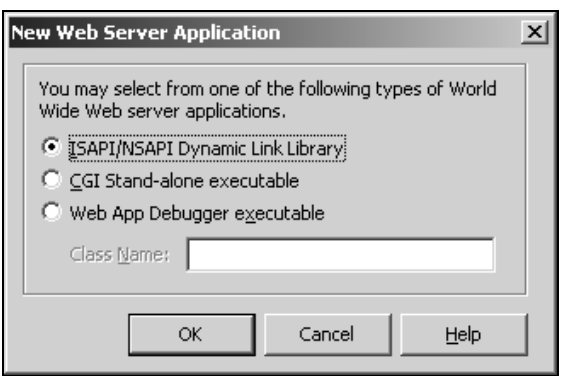


Рис. 37.2. Диалог выбора типа серверного Web-приложения

Здесь доступны следующие типы приложений.

- **ISAPI/NSAPI Dynamic Link Library** — приложение на основе динамической библиотеки. Обмен данными реализует через интерфейс ISAPI (Internet Services API), который детально рассматривается далее. Такие динамические библиотеки загружаются Web-сервером Internet Information Server (IIS) и работают в его адресном пространстве в многопользовательском режиме, когда для каждого запроса создается собственная нить.

  В Delphi реализован специальный класс `TISAPIApplication`, который обеспечивает работу серверного Web-приложения этого типа.

- **CGI Stand-alone executable** — приложение на основе исполняемого файла. Обмен данными осуществляет через интерфейс CGI (Common Gateway Interface), получающий управляющие команды через переменные окружения приложения. Для каждого запроса создается отдельный экземпляр приложения.

- **Web App Debugger executable** — приложение на основе исполняемого файла. Этот тип приложения взаимодействует с утилитой Web App Debugger, которую можно запустить командой меню **Tools | Web App Debugger** главного окна Delphi. Эта утилита представляет собой модель Web-сервера и заменяет настоящие Web-серверы. Это позволяет ускорить процесс отладки серверных Web-приложений средствами Delphi без использования реальных Web-серверов. Затем готовое приложение легко переносится для работы с реальным сервером. Для этого достаточно включить модули тестового приложения в приложение требуемого типа.

В дальнейшем мы будем рассматривать примеры приложений, работающих с Web App Debugger. Это гарантирует, что приложение будет также работать и через Web-сервер Internet Information Server.

Введение в интерфейсы CGI и ISAPI

Итак, в дальнейшем изложении мы будем ориентироваться на Web-сервер Internet Information Server. Как видно из рассмотренных типов серверных Web-приложений, для этого сервера можно создавать приложения на основе как исполняемых файлов, так и динамических библиотек. Для этого применяются интерфейсы ISAPI/NSAPI и CGI. Дадим им краткую характеристику.

Исторически более ранним (и общепринятым) интерфейсом для создания модулей расширения для Web-серверов был интерфейс CGI (Common Gateway Interface), пришедший в Windows из UNIX. Приложение в этом случае представляет собой обычный исполняемый файл (EXE), получающий от сервера необходимую информацию через переменные окружения. В Windows оба этих обстоятельства являются достаточно неудобными. Во-

первых, помещать и читать информацию в окружение DOS не слишком удобно. Во-вторых, для каждого запроса запускается собственный экземпляр приложения. А каждый раз запускать исполняемый модуль — значит, много потерять в скорости.

Первая проблема решается переходом на WinCGI — вариант CGI, адаптированный для Windows. В нем параметры от Web-сервера к приложению передаются через INI-файл. Вторая же проблема остается основным недостатком этого интерфейса.

Излюбленным средством написания скриптов CGI является специальный язык Perl. Конечно, кроме него такие приложения можно писать и на C, Basic и, естественно, в Delphi.

Но есть принципиальный путь ускорения работы Web-приложений. Microsoft снабдила свой Web-сервер Internet Information Server интерфейсом ISAPI (Internet Services API). Он обеспечивает поддержку приложений, представляющих собой динамические библиотеки, загружаемые сервером и работающие с ним в одном адресном пространстве. Одна библиотека может обрабатывать сколь угодно много запросов.

Разновидность интерфейса NSAPI (Netscape Services API) обеспечивает работоспособность приложений при работе с Web-сервером Netscape.

Например, следующий запрос из браузера клиента:

```
http://www.mysite.com/scripts/test.dll/find=someuser
```

означает, что пользователь хочет обратиться на сервер **www.mysite.com** к серверному Web-приложению test.dll с запросом `find=someuser`.

Библиотека ISAPI/NSAPI должна экспортировать вполне определенные функции, которые и будут вызываться сервером. Их три и их объявление в файле проекта ISAPI DLL выглядит следующим образом:

```
exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;
```

В Delphi эти функции объявлены так.

Функция

```
function GetExtensionVersion(var Ver: THSE_VERSION_INFO): BOOL;
```

возвращает в параметре `Ver` номер версии приложения ISAPI или NSAPI.

Функция

```
function HttpExtensionProc(var ECB: TEXTENSION_CONTROL_BLOCK): DWORD;
```

обеспечивает передачу запроса от Web-сервера приложению.

И функция

```
function TerminateExtension(dwFlags: DWORD): BOOL;
```

возвращает `True`, если приложение готово к закрытию.

Как работает серверное Web-приложение

Обобщая изложенное, серверное Web-приложение должно уметь:

- взаимодействовать с Web-сервером, получая от него запросы и возвращая ответы;
- управлять обработкой запросов, применяя необходимые правила бизнес-логики и при необходимости обращаясь за информацией к хранилищам данных.

Когда вы при помощи Репозитория Delphi создаете новый проект серверного Web-приложения, для него, так же как и для обычного приложения, автоматически создается *глобальный объект приложения*. В стандартном приложении это объект `Application` — экземпляр класса `TApplication`.

Все основные функции серверных Web-приложений реализованы в классе `TWebApplication` и его потомках, специализирующихся на конкретных типах приложений. Объекты, созданные на основе этих классов, по выполняемым функциям аналогичны объекту приложения `Application` для обычных приложений Delphi с учетом специфики серверных Web-приложений. Кстати, экземпляр класса серверного Web-приложения также доступен через глобальную переменную `Application`.

В зависимости от используемой технологии разработки структура проекта серверного Web-приложения может несколько отличаться. Например, в технологии WebBroker все важнейшие операции приложения реализованы в Web-модуле *(см. главу 38)*. А в технологии WebSnap функциональность приложения обеспечивают специализированные компоненты *(см. главу 41)*.

Но независимо от типа приложения и выбранной технологии разработки для реализации конкретных функций приложения (бизнес-логики) служат *действия* (actions). Клиентские приложения включают в текст запроса требуемое действие, и в результате серверное приложение, обрабатывая запрос, знает, какую именно операцию необходимо выполнить для создания ответа.

Для каждого нового запроса приложение создает отдельную нить. В зависимости от типа приложения (динамическая библиотека или исполняемый файл) этот процесс имеет особенности.

С точки зрения приложения каждый запрос и ответ — это экземпляр соответствующего класса, которые должно уметь создавать приложение.

Итак, мы можем выделить основные программные части, обеспечивающие работоспособность серверного Web-приложения:

- объект приложения, обеспечивающий прием запросов и передачу ответов Web-серверу;
- объекты запросов и ответов;
- действия, представляющие собой механизм реализации бизнес-логики и управления запросами;
- модули данных, обеспечивающие функционирование компонентов доступа к данным для взаимодействия с хранилищами данных.

Рассмотрим их более детально.

Классы приложений

Исходя из структуры и схемы взаимодействия приложения с Web-сервером *(см. рис. 37.1)*, очевидно, что должен существовать программный код, обеспечивающий получение запросов и отправку ответов, и общее управление частями приложения. Для выполнения этих задач каждое серверное Web-приложение содержит глобальный объект приложения.

Этот объект включается в состав приложения автоматически при его создании и создается при запуске приложения. Доступ к нему осуществляется при помощи глобальной переменной `Application`, подобно обычным приложениям Delphi.

Для каждого типа серверного Web-приложения имеется собственный класс, а их общим предком стал класс `TWebApplication` (рис. 37.3).

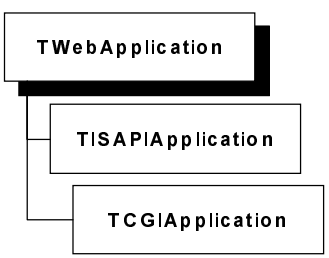


Рис. 37.3. Иерархия классов серверных Web-приложений

Глобальный объект приложения выполняет следующие функции:

- взаимодействие с Web-сервером, получение запросов, отправка ответов;
- создание и уничтожение объектов запросов и ответов, заполнение их свойств;

- управление памятью и пулом Web-модулей (для приложений CGI управление сводится к созданию и удалению экземпляра приложения);
- направление запросов диспетчерам.

Процесс создания и инициализации глобального объекта зависит от типа приложения.

Рассмотрим работу глобального объекта на примере приложения ISAPI.

После получения запроса объект приложения ISAPI создает новую нить и в ней создает экземпляр Web-модуля или же использует существующий экземпляр. Затем создаются объекты запроса и ответа и заполняются их параметры. После этого объект запроса направляется диспетчеру (это может быть Web-модуль или компонент `TWebDispatcher`). В соответствии с параметрами запроса диспетчер направляет его для обработки тому или иному действию для создания ответа. И в конце "технологической цепочки" готовый ответ направляется Web-серверу, а экземпляр Web-модуля уничтожается или кэшируется для дальнейшего использования.

Дополнительно глобальный объект позволяет получать информацию о текущей сессии приложения.

Свойство только для чтения

```
property ActiveCount: Integer;
```

возвращает число запросов, обрабатываемых в настоящее время.

Если предполагается применять Web-модули в пуле без удаления, то свойству

```
property CacheConnections: Boolean;
```

необходимо присвоить значение `True`. И если пулинг Web-модулей включен, то свойство

```
property InactiveCount: Integer;
```

будет возвращать число модулей, зарезервированных в настоящий момент.

Свойство

```
property MaxConnections: Integer;
```

задает максимально возможное число одновременно обрабатываемых запросов (размер пула Web-модулей). Число `ActiveCount` не может превышать число `MaxConnections`. При попытке создать Web-модуль свыше `MaxConnections` генерируется исключительная ситуация.

Действия

Серверные Web-приложения предназначены для выполнения набора операций (правил бизнес-логики) в интересах клиентских приложений. Эти опе-

рации реализуются при помощи коллекций действий. Разработчик серверного Web-приложения создает необходимое число объектов действий и программирует их "начинку" — реализует правила бизнес-логики в методах-обработчиках. Когда клиентское приложение хочет воспользоваться какой-либо функцией сервера, то в запросе он передает название действия и необходимые параметры. Серверное Web-приложение анализирует запрос и направляет его нужному действию.

Функциональность отдельного действия реализована в свойствах и методах класса `TWebActionItem`.

Прежде всего, действие должно быть идентифицировано и связано с запросом. Для этого путь URI запроса необходимо сравнить со свойством действия

```
property PathInfo: string;
```

Если они совпадают, запрос предназначен именно для этого действия.

Примечание

Класс запроса `TWebRequest` имеет аналогичное свойство, которое заполняется автоматически из строки URI при создании.

Для реализации операции, предусмотренной действием, предназначен метод-обработчик

```
type
  THTTPMethodEvent = procedure (Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean) of object;
property OnAction: THTTPMethodEvent;
```

Здесь должен быть подготовлен ответ приложения на данный запрос. Если запрос обработать невозможно или делать этого не следует, параметр `Handled` устанавливается в `False`. В этом случае запрос перенаправляется действию, свойство

```
property Default: Boolean;
```

которого имеет значение `True`.

Процесс поиска действия и выполнения его метода-обработчика `OnAction` выполняется Web-модулем автоматически.

Протоколом HTTP установлено несколько видов запросов от клиента к серверу: GET, POST, HEAD и PUT. Им соответствует свойство действия

```
type TMethodType = (mtAny, mtGet, mtPut, mtPost, mtHead);
property MethodType: TMethodType;
```

Если данное действие может ответить на любой вид запроса, свойство устанавливается в `mtAny`.

При необходимости действие можно включить и отключить. Для этого есть свойство

```
property Enabled: Boolean;
```

Если оно имеет значение `False`, диспетчер запросов пропускает его при поиске кандидата на обработку запроса.

Все необходимые для выполнения функции приложения операции разработчик должен реализовать в методе-обработчике

```
type
  THTTPMethodEvent = procedure (Sender: TObject; Request:
  TWebRequest; Response: TWebResponse; var Handled: Boolean)
  of object;
property OnAction: THTTPMethodEvent;
```

Это событие вызывается при вызове действия объектом диспетчера приложения.

Запросы и ответы

При поступлении запроса от Web-сервера глобальный объект приложения создает объекты запроса и ответа.

Класс `TWebRequest` инкапсулирует запрос, а `TWebResponse` — ответ. Это базовые классы. Но в зависимости от типа серверного Web-приложения в работе используются потомки этих классов для конкретных типов.

Класс запроса содержит большой набор свойств, определяющих параметры заголовка и содержания запроса. Рассмотрим важнейшие из них.

Во первых, Universal Resource Identifier (URI) запроса содержится в свойстве

```
property URL: string;
```

Но кроме этого имеется набор свойств, содержащих составные части URI.

Свойство

```
property Host: string;
```

возвращает имя хоста Web-сервера.

Свойство

```
property ScriptName: string;
```

определяет имя серверного Web-приложения, скрипта или модуля расширения.

Свойство

```
property PathInfo: string;
```

содержит путь URI, который в приложении будет идентифицировать имя действия, предназначенного для выполнения запроса.

Свойство

```
property Query: string;
```

содержит текст запроса с перечисленными в нем параметрами и их значениями.

Но при этом более удобным является свойство

```
property QueryFields: TStrings;
```

которое представляет собой список строк, каждая из которых содержит один параметр запроса в формате `Name=Value`.

Объект запроса возвращает также ряд атрибутов клиента, пославшего запрос. Например, свойство

```
property RemoteAddr: string;
```

содержит IP адрес хоста клиента.

Тип запроса определяется свойством

```
type TMethodType = (mtAny, mtGet, mtPut, mtPost, mtHead);
property MethodType: TMethodType;
```

так же, как и действие. Более того, при направлении запроса тому или иному действию диспетчер сравнивает для них значение этого свойства.

Свойство

```
property Content: string;
```

представляет собой тело или содержание запроса. А в свойстве

```
property ContentFields: TStrings;
```

содержится список параметров и их значений для тела запроса.

Так как запросы и ответы создаются парами, и для объекта запроса всегда существует объект ответа, то класс `TWebResponse` имеет свойство

```
property HTTPRequest: TWebRequest;
```

которое содержит ссылку на объект связанного запроса.

Результат выполнения запроса помещается в свойство

```
property Content: string;
```

объекта ответа.

Модули данных

Одна из задач серверного Web-приложения — взаимодействие с сервером БД для получения данных, необходимых для формирования страниц ответов. Для этого в состав проекта приложения можно добавлять обычные мо-

дули данных. Расположенные на них и правильно настроенные компоненты доступа к данным используются в действиях и полученные ими наборы данных применяются в формировании ответов.

Использование отладочного сервера Web App Debugger

Один из доступных разработчику типов серверного Web-приложения Web App Debugger executable не является стандартным для Internet и применяется исключительно в среде разработке Delphi. Он создан и поддерживается для того, чтобы облегчить процесс разработки Web-приложений. Ведь, например, для разработки приложения ISAPI вам потребуется Web-сервер Internet Information Server. И даже если тестовый IIS установлен на вашем компьютере, для запуска каждой новой версии приложения вам придется перезапускать Web-сервер. Кроме того, для запуска приложения придется переключиться в окно Web-браузера.

Приложение типа Web App Debugger executable позволяет избежать всех перечисленных сложностей на этапе разработки. При создании проекта вы выбираете этот тип приложения и затем проводите полномасштабную разработку проекта — пишете исходный код, настраиваете компоненты, создаете страницы, производите отладку. Легкость работы обеспечивается специальной утилитой WebApp Debugger, которая играет роль Web-сервера для вашего приложения. Утилита обеспечивает полноценный запуск приложения: выбирайте приложение в окне утилиты и после его старта выискивайте ошибки и недочеты. После того как работа завершена, вы можете легко преобразовать ваше приложение к любому другому типу — тому, который будет совместим с коммерческим Web-сервером.

Итак, при создании проекта (см. рис. 37.2) в окне **New WebServer Application** вам необходимо выбрать радиокнопку **Web App Debugger executable** и затем в строке **Class Name** задать имя приложения. Это всего лишь название, которым будет именоваться ваше приложение в списке приложений утилиты Web App Debugger.

После сохранения и компиляции проекта приложение доступно для выполнения и отладки. Для этого необходимо запустить утилиту WebApp Debugger (рис. 37.4) при помощи одноименного пункта из меню **Tools** главного окна Delphi. Если сервер еще не запущен, это необходимо сделать при помощи кнопки **Start**. Затем выберите ссылку **Default URL** и затем в развернувшемся окне Web-браузера — ссылку **View Details**. Из появившегося списка выберите и запустите тестовое приложение — появится главная страница приложения.

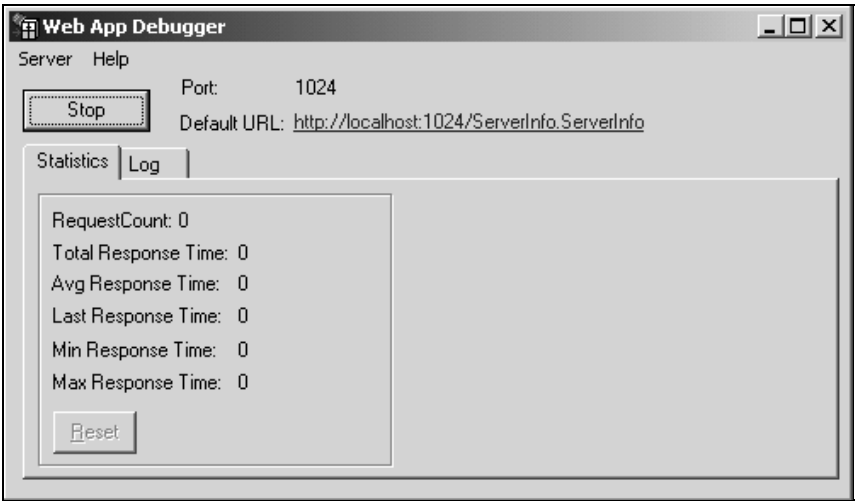


Рис. 37.4. Утилита Web App Debugger для отладки серверных Web-приложений

Перенос тестового приложения на рабочую платформу

Тип приложения Web App Debugger executable позволяет существенно ускорить процесс отладки. Но для распространения этот тип, конечно же, не подходит. Поэтому опишем, например, простой процесс переноса приложения на платформу ISAPI.

1. Создайте новый проект серверного Web-приложения и сохраните его в отдельной папке с теми же именами файлов, что и для тестового приложения. При этом не забудьте также изменить имена классов Web-модулей.
2. Перенесите из папки тестового проекта все созданные модули (в нашем случае это только uMain.pas).
3. При необходимости включите в проект дополнительные функции (например, уничтожения экземпляра приложения после заданного интервала простоя).
4. Откомпилируйте проект.

На этом все. Приложение ISAPI готово к работе.

Особенности отладки приложений ISAPI

Как и любое другое приложение, динамические библиотеки ISAPI нуждаются в отладке. Но это необычные приложения, и здесь есть много тонкостей,

которые должны быть известны любому Web-программисту, работающему в Delphi.

Первые проблемы возникают еще до отладки — вы можете столкнуться с тем, что не удается заменить старый файл DLL в виртуальной папке IIS, где он исполняется, или даже построить измененный вариант проекта (если вы компилируете прямо в эту виртуальную папку).

Дело в том, что расширения ISAPI кэшируются всеми версиями Microsoft IIS. Поэтому заменить ISAPI DLL, которая уже запускалась сервером, невозможно, т. к. она в данный момент отображена в память и, следовательно, заблокирована системой. Для замены нужно выбрать один из следующих путей.

1. Если виртуальная папка, где находится ваше ISAPI-расширение, имеет уровень безопасности Low или Medium, то придется каждый раз перезапускать службу IIS. При этом кэш приложений сбрасывается.

   Процедура эта небыстрая и даже на мощном компьютере занимает до десяти секунд. К тому же это невозможно, если вы отлаживаетесь на "реальном" сервере, который посещается пользователями.

2. Если виртуальная папка, где находится ваше ISAPI-расширение, имеет уровень безопасности High (Isolated), то для перекомпиляции достаточно завершить тот экземпляр приложения dllhost.exe, в котором оно запускается. Как отличить в Диспетчере задач нужный экземпляр dllhost.exe, ведь их будет как минимум два (а если есть и другие изолированные ISAPI-приложения, то и больше)? В этом случае откройте Диспетчер задач и, наблюдая за ним, вызовите из браузера ваше приложение. Появившийся при вызове экземпляр dllhost и будет искомым. Запомните его `Process ID` и при необходимости завершите.

3. Наконец, самый радикальный метод состоит в отключении кэширования ISAPI для Web-сервера. Для IIS 5 это можно сделать в диалоге **Home Directory | Configuration** в свойствах всего Web-узла. Для более старых версий (где эта возможность отсутствует) может понадобиться войти в редактор системного реестра и изменить там значение ключа `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters\CacheExtensions` с 1 на 0, и после этого перезапустить сервер. Кэширование отключится, и вы сможете перезаписывать файл ISAPI DLL на диске при необходимости. Важно учитывать, что это влечет гигантское снижение производительности сервера, не забудьте снова включить кэширование, когда все ваши приложения отлажены.

Теперь непосредственно об отладке. "Напрямую" отлаживать модули ISAPI нельзя, т. к. это динамические библиотеки и они нуждаются в запускающем приложении. В Delphi реализована возможность отладки динамических библиотек, запускаемых другими EXE-файлами. Проблема здесь лишь в том,

что IIS — это системная служба. Методика отладки ISAPI-модулей в рамках процесса IIS 5 (inetinfo.exe) описана в статье на сайте Borland Community: **http://community.borland.com/article/0,1410,23024,00.html**. Там же можно найти аналогичный материал для версии PWS (Personal Web Services) для Windows 98.

Более изящный подход изложен на сайте **www.msdelphi.com** в статье "Debugging IIS5 the easy way". Отлаживаясь в процессе IIS, вы подвергаете его риску и дополнительным нагрузкам. Описываемый ниже подход лишен этого недостатка.

1. Первым делом нужно создать виртуальную папку IIS, в которой будут находиться отлаживаемые приложения. Назовем ее, скажем, `ISAPIDebug`. Для чего это нам нужно? Для того чтобы установить для нее уровень безопасности High. Как мы уже писали, в этом случае все приложения в этой папке будут исполняться в контексте отдельного процесса dllhost.exe. Эта системная утилита представляет собой приложение COM+. После создания виртуальной папки зайдите в оснастку Component Services (Control Panel/ Administrative Tools), которая отвечает за настройку приложений COM+. Вы увидите там новое приложение, соответствующее вашей папке. На рис. 37.5 оно называется IIS-{Default Web Site//Root/ISAPIDebug}.

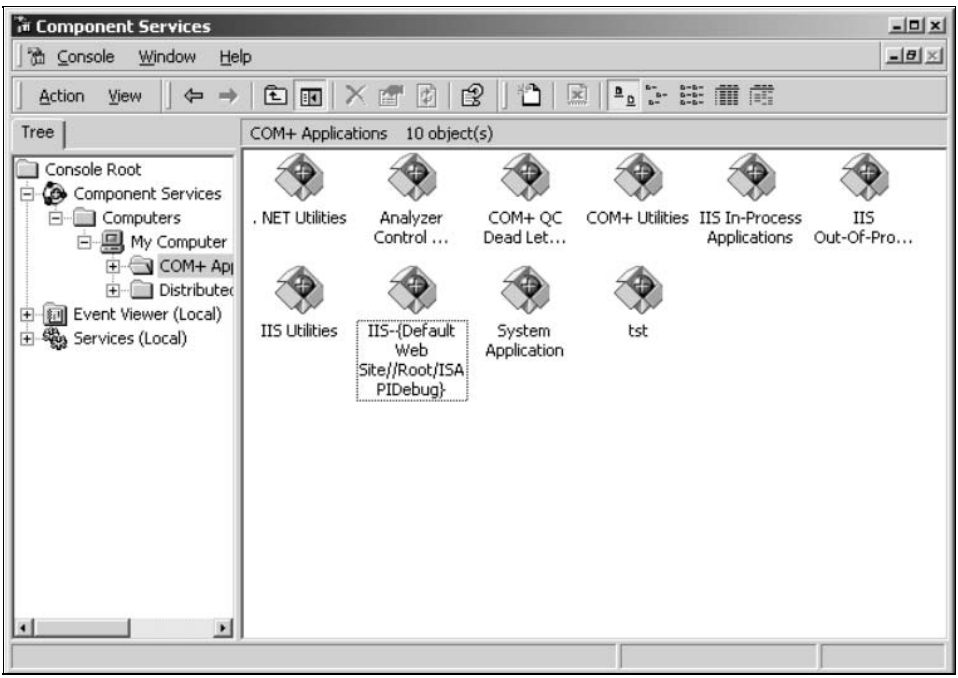


Рис. 37.5. Этапы настройки ISAPI-приложения для отладки в Delphi: оснастка настройки приложений COM+

2. Откройте диалоговое окно свойств этого приложения. Здесь нам понадобятся две вещи: во-первых, нужно запомнить (а точнее, записать в буфер обмена) ID приложения (рис. 37.6 — Application ID, равный в нашем случае {DB431D8F-292D-4EC8-BD37-054C1BB6378D}). Во-вторых, следует изменить настройки идентификации по умолчанию. На странице **Identity** этого диалога нужно изменить учетную запись пользователя, от имени которого запускается приложение. Для того чтобы избежать проблем с правами доступа на отладку, установите учетную запись текущего интерактивного пользователя (рис. 37.7).

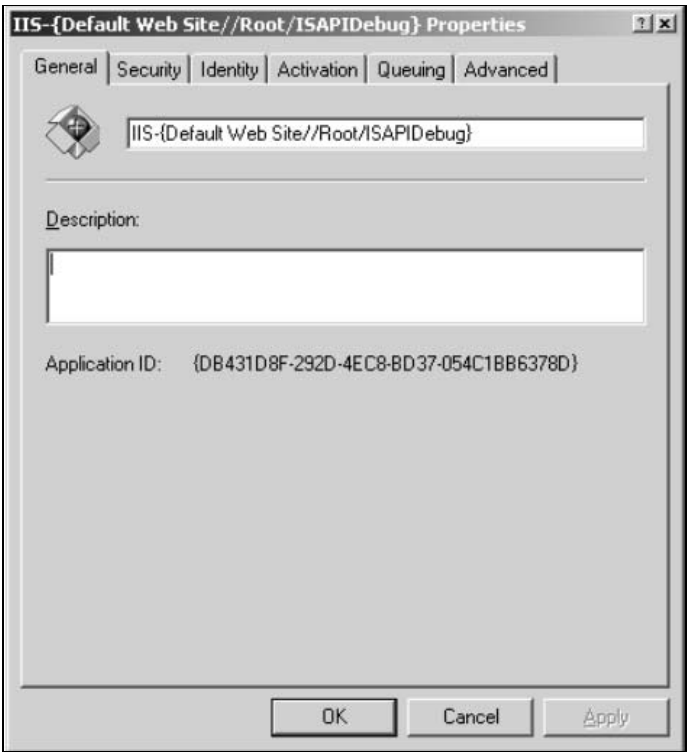


Рис. 37.6. Этапы настройки ISAPI-приложения для отладки в Delphi: диалоговое окно с указанием Application ID нужного экземпляра dllhost.exe

3. На этом настройка dllhost закончена. Для установки параметров в Delphi нам понадобится Application ID, который к этому моменту должен быть у вас в буфере обмена. Параметры запуска вашего приложения в Delphi выглядят, как показано на рис. 37.8 (разумеется, если путь к системным папкам у вас отличается от указанного, замените значение параметра **Host Application**).

Рис. 37.7. Этапы настройки ISAPI-приложения для отладки в Delphi: настройка идентификации

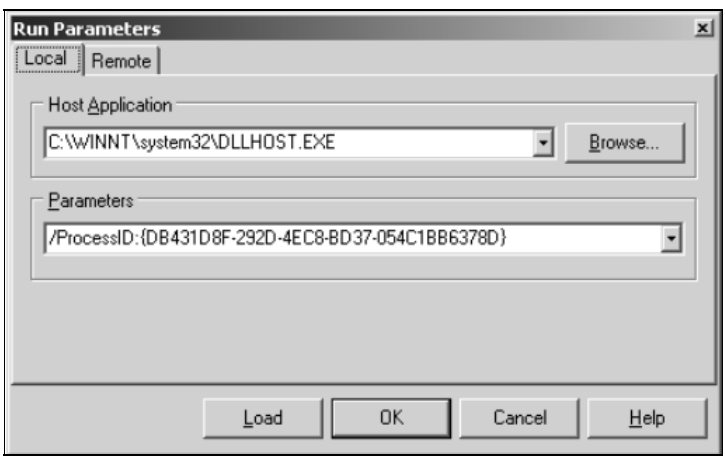


Рис. 37.8. Этапы настройки ISAPI-приложения для отладки в Delphi: настройка параметров запуска приложения в среде Delphi

4. После перезапуска IIS ваше приложение готово к отладке. Нужно лишь иметь в виду то, что вам придется компилировать проект прямо в папку ISAPIDebug, иначе нужный файл библиотеки не отладить. Для запуска, как обычно, нажмите клавишу <F9> и... ничего не случится. Фактический запуск произойдет тогда, когда IIS запустит ваш экземпляр dllhost, а тот загрузит вашу DLL. Создайте окно браузера и наберите в нем URL, например: http://localhost/ISAPIDebug/SimpleWebApp.dll. Вот тогда-то среда Delphi и сможет получить управление для целей отладки.

Резюме

Распределенные Web-приложения используют в качестве транспортной среды глобальные возможности коммуникации Internet. Web-серверы обеспечивают управление клиентскими запросами, а сами клиентские приложения — это страницы HTML в стандартных Web-браузерах.

Серверные Web-приложения — важнейшая составная часть распределенных Web-приложений. Такое приложение обеспечивает обработку запросов клиентов, которые поступают к нему от Web-сервера. Серверное Web-приложение получает запрос, обрабатывает его в соответствии с правилами бизнес-логики, при необходимости обращается к серверу БД, формирует ответ и оформляет его с помощью шаблонов страниц.

Важной особенностью таких приложений является динамическая генерация страниц ответов. Для этого применяются шаблоны страниц, которые служат для создания страниц HTML ответов.

Глава 38

Технология Web Broker

Начиная с этой и в последующих главах данной части мы расскажем о различных методиках разработки серверных Web-приложений — основной части распределенного Web-приложения. Роль серверного Web-приложения заключается в получении и обработке клиентских запросов и создании ответов на эти запросы. В нем сосредоточена основная функциональность распределенного приложения. Другие же части обеспечивают использование Internet как транспортного окружения для распределенного приложения. Это Web-сервер, который осуществляет соединение серверного Web-приложения с клиентами и передачу им страниц в формате HTML при помощи протокола HTTP. Для просмотра страниц клиенты применяют обычные Web-браузеры. *(Подробнее об архитектуре распределенного Web-приложения см. главу 37.)*

В чем же заключаются основные задачи серверного Web-приложения? Во-первых, за неимением других подходящих мест оно просто обязано содержать бизнес-логику распределенного приложения. Во-вторых, оно должно уметь взаимодействовать с Web-сервером. Для этого ему необходимо иметь механизмы для обработки и диспетчеризации запросов. Каждый запрос должен быть распознан и направлен для создания ответа соответствующей странице или модулю, инкапсулирующему бизнес-логику приложения. В-третьих, серверное Web-приложение должно содержать механизм динамической генерации страниц HTML, представляющих собой ответы на запросы клиентов.

Для реализации перечисленных возможностей в Delphi имеется несколько технологий. В этой главе мы рассмотрим технологию *Web Broker*, которая объединяет набор компонентов, модулей и мастеров.

Знание HTML для изучения этой и последующих глав пригодится, но только в объеме, достаточном для увязки непосредственно с вопросами программирования, а художественное оформление вашего продукта лучше оставить профессиональному Web-дизайнеру. Наряду с тегами HTML для создания Web-страниц серверных приложений пригоден и язык XML, варианты работы с которым в Delphi рассматривались в предыдущих главах.

В этой главе рассматриваются следующие вопросы.

- Структура серверного Web-приложения на основе технологии Web Broker.
- Компоненты-брокеры и компоненты-продюсеры.
- Использование языка HTML в приложениях Delphi.
- Создание и оформление форм ввода данных.
- Компоненты доступа к БД.
- Файлы cookies.
- Пример создания серверного Web-приложения.

Обзор технологии Web Broker

Технология создания распределенных Web-приложений Web Broker впервые появилась в Delphi 5 и с тех пор не претерпела существенных изменений. Конечно же, за прошедшее время появились новые, более совершенные средства разработки. Взять, к примеру, технологию WebSnap, которая появилась в Delphi 6 и является прямым потомком Web Broker, добавив к базовой технологии усовершенствованную структуру приложений и возможность использования скриптов в создаваемых страницах. Тем не менее основные механизмы и методики остались неизменными, и их изучением мы и займемся в этой главе. А в целом технология Web Broker более проста для понимания и применения.

Итак, технология Web Broker позволяет разрабатывать серверные Web-приложения трех типов, создавая исполняемые файлы или динамические библиотеки *(см. главу 37)*. Приложения на основе Web Broker умеют взаимодействовать с Web-сервером и осуществлять диспетчеризацию запросов пользователей. Для обработки каждого нового запроса создается отдельная нить, которая после окончания работы может не удаляться, а кэшироваться для обработки других запросов.

Несколько слов о страницах ответов. Приложения Web Broker могут создавать страницы ответов на языках HTML и XML. При этом можно применять как заранее созданные страницы, так и шаблоны — заготовки страниц, которые динамически трансформируются в готовые страницы добавлением к ним данных, необходимых для ответа. Динамическая генерация, безусловно, предпочтительнее, т. к. позволяет гибко реагировать на весьма вероятные изменения в оформлении страниц и избавляет от хранения сотен и тысяч статических страниц HTML.

За динамическую генерацию страниц отвечает семейство компонентов-продюсеров. Они позволяют создавать страница на основе не только HTML-

заготовок, но и наборов данных, которые серверное Web-приложение может получать таким же образом, как и любое приложение БД.

В рамках технологии Web Broker нельзя работать со скриптами VBScript или JScript.

Структура серверного Web-приложения на основе технологии Web Broker

Итак, серверное Web-приложение взаимодействует с Web-сервером, получая от него запросы и создавая ответы на основе правил бизнес-логики. Давайте посмотрим, что же происходит при этом внутри приложения, и какие компоненты Web Broker в этом участвуют (рис. 38.1).

Во-первых, с точки зрения функциональности, серверное Web-приложение состоит из трех взаимосвязанных частей, выполняющих следующие операции.

- Диспетчеризация запросов. Приложение должно уметь обработать запрос и получить из него информацию, какая именно Web-страница должна послужить основой для создания ответа на запрос. Функции диспетчеризации инкапсулированы Web-модулем, который представляет собой отдаленного потомка класса `TDataModule`, и специализированным компонентом `TWebDispatcher`.
- Реализация бизнес-логики. Приложение должно уметь делать то, для чего собственно и создавалось. В ответ на поступивший и правильно обработанный запрос приложение должно вызвать некую заранее созданную и отлаженную процедуру, реализующую нужную клиенту операцию. Это может быть и простое нажатие кнопки, обеспечивающее переход на следующую страницу, и сохранение финансовой транзакции, и подтверждение заказа авиабилета, и т. д.

  Для реализации бизнес-логики служат специальные объекты — действия, методы-обработчики которых могут быть вызваны при обработке запроса. Именно в них и надлежит реализовывать необходимые приложению операции. Действия инкапсулированы в Web-модулях.
- Формирование Web-страниц. В большинстве случаев серверное приложение предоставляет клиентам пользовательский интерфейс, реализованный на основе Web-страниц. При этом наиболее предпочтителен динамический способ их формирования с помощью *компонентов-продюсеров*, которые на основе *шаблонов* HTML или XML умеют формировать страницу, т. е. создавать набор тегов HTML и элементов XML, обеспечивающих корректное представление и поведение страницы в Web-браузере клиента.

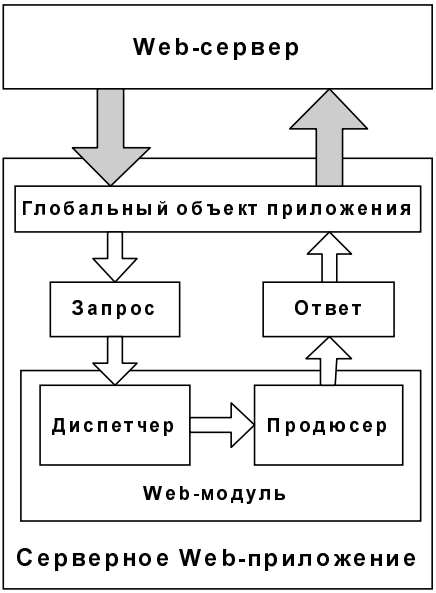


Рис. 38.1. Структура серверного Web-приложения на основе технологии Web Broker

С точки зрения разработчика серверное Web-приложение, созданное с помощью технологии Web Broker, состоит из следующих частей.

Глобальный объект приложения инкапсулирует функции взаимодействия приложения с Web-сервером и обеспечивает получение запросов и отправку готовых ответов. После получения запроса он создает новую нить и направляет запрос диспетчеру Web-модуля для обработки.

Основу приложения составляют *Web-модули* — объекты, подобные модулям данных, но обладающие дополнительными функциями. Именно Web-модули обеспечивают диспетчеризацию запросов. Кроме этого, они инкапсулируют действия. Впрочем, они прекрасно справляются и с традиционной ролью — быть площадкой для хранения невизуальных компонентов и их программного кода.

И наконец, *компоненты-продюсеры* обеспечивают динамическую генерацию Web-страниц. Для этого они могут задействовать шаблоны HTML или XML или таблицы баз данных. В категории **Internet** Палитры инструментов доступны несколько компонентов, обеспечивающих доступ к разным источникам данных.

Теперь рассмотрим перечисленные составные части серверного Web-приложения более подробно.

Создание серверного Web-приложения

Для создания серверного Web-приложения откройте диалоговое окно Репозитория Delphi (команда **File | New | Other** главного меню) и в нем в категории **Delphi Projects | WebBroker** щелкните на значке **Web Server Application**. В результате появится окно **New Web Server Application**, в котором необходимо выбрать тип будущего приложения (см. рис. 37.2). Подробно типы серверных Web-приложений описаны в *главе 37*.

После того как вы выбрали тип приложения и нажали кнопку **OK**, среда разработки создаст новый проект со всеми необходимыми составными частями. Если вы сразу после создания откомпилируете проект, то получите серверное Web-приложение с полностью отсутствующей бизнес-логикой, но, тем не менее, умеющее взаимодействовать с Web-сервером и обрабатывать запросы. Базовая функциональность серверного Web-приложения по диспетчеризации запросов реализована в глобальном объекте приложения и Web-модуле, который добавляется к проекту автоматически при создании. (Подробнее об этом см. далее.) При этом глобальный объект приложения берет на себя взаимодействие с Web-сервером, а Web-модуль обеспечивает обработку поступивших запросов. Причем первый выполняет свою работу автоматически, а для второго необходимо дополнительное программирование.

Помимо традиционной задачи — быть хранилищем невизуальных компонентов и правил бизнес-логики — Web-модуль может использовать механизм действий. Web-модуль имеет свойство `Actions`, представляющее собой индексированный список для объектов класса `TWebActionItem`. То или иное действие выполняется, если пришедший запрос имеет параметр, совпадающий с именем действия. В этом случае вызывается метод-обработчик `OnAction` действия. И именно в этом методе-обработчике вы должны разместить необходимые бизнес-правила. В результате клиент может заставить серверное Web-приложение выполнять вполне определенные операции.

И о клиентских приложениях. Согласно материалу *главы 37*, распределенное Web-приложение на стороне клиента использует Web-браузер для просмотра страниц ответов в формате HTML, присланных сервером. Крайне желательно, чтобы серверное Web-приложение генерировало страницы ответов динамически, а не извлекало из необъятных хранилищ. Для этого применяются специализированные компоненты-продюсеры, расположенные в категории **Internet** Палитры инструментов.

Компоненты-продюсеры размещаются в Web-модуле, настраиваются и связываются с нужными действиями. В результате в ответ на конкретный запрос клиента будет активизировано некоторое действие, в методе-обработчике которого расположен код, обеспечивающий генерацию страни-

цы ответа компонентом-продюсером. И в Web-браузер клиента будет переслана правильная страница ответа.

Такова вкратце методология разработки серверных Web-приложений. А теперь остановимся на его составных частях более детально.

Диспетчеризация запросов. Глобальный объект приложения

Серверное Web-приложение, созданное при помощи технологии Web Broker, включает глобальный объект приложения *(см. главу 37)*, который обеспечивает взаимодействие приложения с Web-сервером. Доступ к нему, так же как и в обычном приложении, осуществляется при помощи глобальной переменной `Application`.

Для каждого типа серверного Web-приложения имеется собственный класс, а их общим предком стал класс `TWebApplication`.

И в зависимости от типа классы серверного приложения имеют отличия в функциональности.

Приложения CGI представляют собой исполняемые файлы. Поэтому класс `TCGIApplication` для таких приложений осуществляет обработку каждого поступающего запроса методом `Run`.

Приложения ISAPI/NSAPI представляют собой динамические библиотеки. Поэтому класс `TISAPIApplication` содержит три метода, соответствующих методам DLL приложения: `GetExtensionVersion`, `HttpExtensionProc` и `TerminateExtension`.

Как видите, функциональность каждого класса жестко привязана к типу приложения. При разработке собственного серверного Web-приложения вам вряд ли понадобится работать с этими методами напрямую. Но некоторые свойства глобального объекта приложения могут пригодиться. Например, с их помощью можно получить информацию о сессии, обеспечить управление Web-модулем и т. д. Более подробно об этом *см. главу 37*.

Web-модули и действия

Основа проекта в технологии Web Broker — Web-модуль, представленный классом `TWebModule`. На нем, как на фундаменте, стоит любое из типов серверных Web-приложений. Это то звено, которое принимает запросы, интерпретирует их и направляет для выполнения.

Для этого класс Web-модуля наследует функциональность базового диспетчера запросов — класса `TCustomWebDispatcher`. Аналогичными возможно-

стями обладает и компонент `TWebDispatcher`. И благодаря этому разработчики могут создавать собственные диспетчеры запросов.

Примечание

Попытка поместить компонент `TWebDispatcher` на уже открытый Web-модуль приведет к появлению сообщения об ошибке "Only one dispatcher for module".

Кроме функции диспетчеризации запросов Web-модуль позволяет программировать бизнес-логику приложения.

Свойство

```
property Actions: TWebActionItems;
```

как раз и содержит индексированный список объектов действий, играющих роль обработчиков запросов. Каждое из них идентифицируется текстовой строкой — свойством `TWebActionItem.PathInfo`. Так, на запрос

```
http://www.mysite.com/scripts/test.dll/add
```

будет вызвано действие, свойство `PathInfo` которого равно `add`.

Для работы со списком действий предусмотрен редактор свойства `Actions` (рис. 38.2). Помимо стандартных для такого рода редакторов свойств на основе индексированных списков, данный редактор позволяет увидеть для каждого действия ряд важнейших свойств, в том числе свойство `PathInfo`.

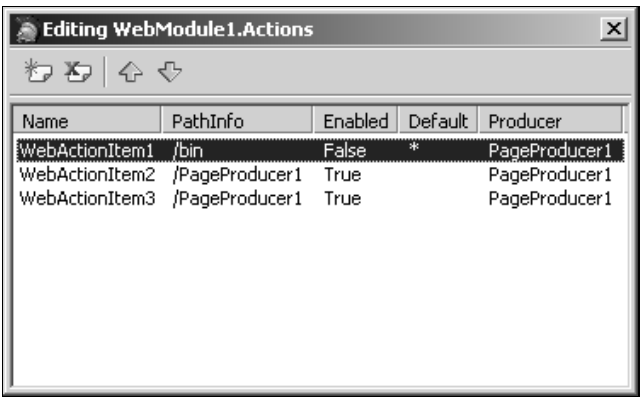


Рис. 38.2. Редактор действий Web-модуля

Для обработки запросов, для которых не найдено соответствующее имя действия, Web-модуль может назначить действие по умолчанию. Это действие, свойство `TWebActionItem.Default` которого установлено в `True`.

Класс TWebModule имеет два события, которые происходят до и после обработки запроса:

```
type
  type THTTPMethodEvent = procedure (Sender: TObject; Request:
  TWebRequest; Response: TWebResponse; var Handled: Boolean) of object;
property BeforeDispatch: THTTPMethodEvent;
property AfterDispatch: THTTPMethodEvent;
```

Здесь вы имеете, соответственно, первую и последнюю возможность что-либо поменять в ответе на запрос или вообще отказаться от его обработки. Если в BeforeDispatch установить параметр Handled в True, на этом обработка запроса заканчивается:

```
procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject; Request:
TWebRequest; Response: TWebResponse; var Handled: Boolean);
var i: Integer;
    IsActionExist: Boolean;
begin
 IsActionExist := False;
 for i := 0 to WebModule1.Actions.Count - 1 do
  if Request.PathInfo = WebModule1.Actions[i].PathInfo
   then IsActionExist := True;
 Handled := IsActionExist;
end;
```

Параметры Request и Response этих методов-обработчиков предоставляют доступ к объектам запроса и ответа соответственно (подробнее об этих объектах см. далее). Но т. к. Web-модуль получает ссылки на созданные глобальным объектом приложения объекты запроса и ответа при своем создании или извлечении из пула, доступ к ним можно получить при помощи свойств

```
property Request: TWebRequest;
property Response: TWebResponse;
```

Специально для того, чтобы упростить диспетчеризацию и обеспечить доступ к действию по имени, существует метод

```
function ActionByName(const AName: string): TWebActionItem;
```

Он возвращает ссылку на элемент Action по его имени.

Разработчик может редактировать список действий Web-модуля как на этапе разработки, так и программными средствами во время выполнения. Для этого существуют методы списка действий Actions:

```
function Add: TWebActionItem;
```

и

```
procedure Delete(Index: Integer);
```

Например:

```
var NewAction: TWebActionItem;
    FAddActionExecute: THTTPMethodEvent;
...
 NewAction := WebModule1.Actions.Add;
 NewAction.Name := 'actAdd';
 NewAction.DisplayName := 'Add Action';
 NewAction.MethodType := mtAny;
 NewAction.PathInfo := 'add';
 NewAction.OnAction := FAddActionExecute;
...
```

События

```
property OnCreate: TNotifyEvent;
property OnDestroy: TNotifyEvent;
```

происходят при создании и уничтожении экземпляра `TWebModule`. Особо нужно обратить внимание на первое событие — обработчик `OnCreate` вызывается еще до обработки первого HTTP-запроса. Именно здесь можно сделать следующее:

- прочитать из INI-файлов или системного реестра текущие настройки сервера;
- инициализировать глобальные переменные;
- подключиться к базе данных.

Компоненты-продюсеры

В состав технологии Web Broker входят компоненты-продюсеры, расположенные в категорию **Internet** Палитры инструментов. Они обеспечивают динамическую генерацию Web-страниц на основе шаблонов HTML или задействуют базы данных в качестве источника данных.

Наиболее общие функции продюсера реализованы в компоненте `TPageProducer`. Остальные компоненты-продюсеры обеспечивают генерацию страниц для отображения данных, получаемых из наборов данных. Сейчас мы подробно рассмотрим общие свойства и методы компонентов-продюсеров на примере компонента `TPageProducer`, а к остальным компонентам обратимся позже.

Во-первых, для динамически создаваемой страницы должен существовать шаблон — основа, заготовка, описывающая общие характеристики и параметры страницы.

Шаблон для создания страницы задается свойством

```
property HTMLDoc: TStrings;
```

Если шаблон хранится в файле, то его можно определить свойством

```
property HTMLFile: TFileName;
```

Для получения страницы из шаблона необходимо вызвать метод

```
function Content: string;
```

но он сработает только, если шаблон задан перечисленными выше свойствами. В противном случае подойдет метод

```
function ContentFromString(const S: string): string;
```

который формирует HTML-текст страницы прямо из строки `S`.

И еще один подобный метод

```
function ContentFromStream(Stream: TStream): string;
```

формирует результат из данных потока `Stream`.

Если шаблон содержит скрипты, их язык должен быть задан свойством

```
property ScriptEngine: string;
```

В Delphi доступны языки скриптов VBScript и Jscript.

Еще одно свойство

```
property Dispatcher: IWebDispatcherAccess;
```

содержит ссылку на Web-модуль или компонент `TWebDispatcher`, который обслуживает текущий запрос.

Единственный метод-обработчик компонента

```
type
  TTag = (tgCustom, tgLink, tgImage, tgTable, tgImageMap, tgObject,
  tgEmbed);
  THTMLTagEvent = procedure (Sender: TObject; Tag: TTag; const TagString:
  string; TagParams: TStrings; var ReplaceText: string) of object;
OnHTMLTag: THTMLTagEvent;
```

вызывается при необходимости интерпретировать тег HTML шаблона, который не может быть обработан интерпретатором HTML. Полный текст тега содержится в параметре `TagString`. Параметры тега содержатся в параметре `TagParams` в формате `Name=Value`. Преобразованный текст должен быть сохранен в параметре `ReplaceText`.

Параметр `Tag` определяет тип встреченного тега. Из семи разновидностей шесть — предопределенные (табл. 38.1).

Таблица 38.1. *Типы тегов для компонента-продюсера*

Значение параметра `Tag`	Текст тега	Назначение
`TgLink`	`LINK`	Описание гипертекстовой ссылки
`TgImage`	`IMAGE`	Описание изображения
`tgTable`	`TABLE`	Описание таблицы
`tgImageMap`	`IMAGEMAP`	Описание изображения с контекстно-чувствительными областями
`tgObject`	`OBJECT`	Описание объекта ActiveX, встраиваемого в страницу
`TgEmbed`	`EMBED`	Описание встраиваемой библиотеки Netscape

Реагируя на один из предопределенных тегов, компонент-продюсер должен сгенерировать и вставить в документ соответствующий правильно описанный элемент HTML. Если же в документе встретился тег, не совпадающий ни с одним из перечисленных, то свойство `Tag` будет равно `tgCustom`. Такие теги вы можете обрабатывать по своему усмотрению.

Например, метод-обработчик

```
procedure TForm1.ppHTMLTag(Sender: TObject; Tag: TTag; const TagString:
String; TagParams: TStrings; var ReplaceText: String);
begin
 if Tag = tgImage then
  if (TagParams.Values['ImgType'] = 'GIF') and
     (TagParams.Values['ImgName'] <> '')
   then ReplaceText := '<IMG SRC="../Images/' +
                       TagParams.Values['ImageName'] + '">';
end;
```

заменяет тег ссылкой на конкретное изображение.

Страницы HTML в серверных Web-приложениях

Теперь перейдем к некоторым особенностям программирования пользовательского интерфейса HTML-страниц серверных Web-приложений. Мы не

будем останавливаться на богатых возможностях языка HTML, для этого имеется широчайший выбор специальной литературы. И в то же время рассказ о страницах с приветствиями вроде "Hello, World!" вряд ли окажется полезным. Поэтому мы рассмотрим только некоторые тонкости компонентов Web Broker, которые помогут в реальной разработке HTML-страниц.

Отображение данных

Если ваше приложение в ответ на запросы генерирует только одну страницу, ответ можно сформировать прямо в методе-обработчике события `OnAction` соответствующего действия:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 PageProducer1.HTMLFile := FPageFileName;
 Response.Content := PageProducer1.Content;
end;
```

Если же на выходе не одна, а множество страниц, необходима автоматизация этого действия. Можно поступить следующим образом. При создании Web-модуля нужно задать типовой набор тегов для заголовков всех страниц, а затем дописывать его в начало и конец каждой создаваемой страницы. Тогда они будут выглядеть похоже. К тому же, таким образом можно провести настройку внешнего вида, прочитав часть параметров из файла инициализации (INI) или реестра (листинг 38.1).

Листинг 38.1. Настройка внешнего вида Web-страниц

```
var CommonLookStart, CommonLookEnd: String;
procedure TWebModule1.WebModule1Create(Sender: TObject);
var Ini: TINIFIle;
    s: string;
begin
 Ini := TINIFile.Create('SomeApp.ini');
 try
  bgpath := Ini.ReadString('Design','Background','img\star.jpg');
  txtcol := Ini.ReadString('Design','text','white');
  lcol := Ini.ReadString('Design','link','lime');
  acol := Ini.ReadString('Design','alink','olive');
  vcol := Ini.ReadString('Design','vlink','green');
 finally
```

```
  ini.Free;
 end;
  CommonLookStart := Format('<HTML><BODY BACKGROUND="%s" TEXT=%s LINK=%s
ALINK=%s VLINK=%s>',
  [bgpath,txtcol,lcol,acol,vcol]);
  CommonLookEnd := '</BODY></HTML>';
end;
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 SomePageProducer.HTMLFile := FPageFileName;
 Response.Content := CommonLookStart +
                     SomePageProducer.Content +
                     CommonLookEnd;
end;
```

Следующая проблема, с которой наверняка придется столкнуться, — изменение содержания страниц в зависимости от некоторых условий. Конечно, можно предусмотреть все варианты прямо в коде события `OnAction` и написать множество операторов `if..then..else`. Но есть и более элегантное решение — реализовать компоненты-продюсеры Web Broker.

Примечание

Данные компоненты могут быть использованы не только в составе ISAPI- или CGI-приложения, но и в любом случае, когда требуется сгенерировать документ в формате HTML.

Пусть, например, необходимо включить в ответ информацию из запроса.

Для начала создадим шаблон HTML. Для этого в свойстве `HTMLDoc` компонента-продюсера сохраним такой набор строк:

```
<HTML><HEAD>
<TITLE>Demo 2</TITLE>
<BODY>
Clients request host: <#NAME>!
Request tiem: <#TIME>
</BODY>
</HTML>
```

Разбору в данном случае подлежат определенные нами теги `<#NAME>` и `<#TIME>`.

У компонента TPageProducer есть свойство

```
property Dispatcher: TCustomWebDispatcher;
```

Оно содержит ссылку на Web-модуль или компонент TWebDispatcher, который обслуживает текущий запрос. Тогда метод-обработчик OnHTMLTag

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
 if TagString='NAME' then
 ReplaceText := TCustomContentProducer
(Sender).Dispatcher.Request.RemoteHost
 else if TagString='TIME' then ReplaceText := TimeToStr(Now);
end;
```

обеспечивает интерпретацию тегов <NAME> и <TIME>, заменяя их текст на IP-адрес хоста запроса и время отправки соответственно. Затем метод-обработчик действия передает текст HTML ответу.

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 Response.Content := PageProducer1.Content;
end;
```

В результате на тестовой странице динамически будет отображаться информация о хосте клиента и времени запроса.

Ввод и редактирование данных

Мы познакомились с тем, как выводить информацию на Web-страницах. Но это только половина дела — нужно еще уметь ввести данные от пользователя. С помощью того же механизма страниц можно создать форму ввода. Для этого существует специальный тег — <FORM> (и соответствующий ему конечный тег </FORM>). В теге укажите URL вашего серверного приложения, которое будет обрабатывать поступившие данные, например:

```
<FORM method="GET" action="http://localhost/script/some.dll/validate>
```

Для ввода информации служит тег <INPUT>. Он специально предназначен для организации диалога с пользователем в среде браузера и имитирует основные элементы управления Windows, предназначенные для ввода информации. В каждом теге должны быть поля type (задающие тип элемента управления) и name. Основные типы элементов управления приведены в табл. 38.2.

Таблица 38.2. *Типы элементов управления Web-страниц (тег <INPUT>)*

Тэг	Назначение
<input type=TEXT>	Однострочный текстовый редактор. Предназначен для ввода обычного текста
<input type=PASSWORD>	Однострочный текстовый редактор для ввода зашифрованной информации (паролей)
<input type=FILE>	Однострочный редактор для выбора файла
<input type=CHECKBOX>	Кнопка с независимой фиксацией (флажок)
<input type=RADIO>	Кнопка с зависимой фиксацией
<input type=HIDDEN>	Нередактируемое поле, содержащее служебную информацию для сервера, скрытую от пользователя
<input type=SELECT>	Кнопка, нажатие которой инициирует передачу информации из браузера на сервер
<input type=RESET>	Кнопка, нажатие которой очищает все поля ввода и устанавливает их в начальное значение

Теперь создайте шаблон страницы, содержащий набор элементов управления для ввода имени пользователя и пароля (листинг 38.2).

Листинг 38.2. Шаблон HTML для создания страницы аутентификации пользователей

```
<form method="GET" action="/stest/some.dll/validate">
<CENTER>
<h2>Login Page</h2>
</center>
<BR>
<h3>Enter user name:</h3>
<table>
<td><input name="username" type="TEXT"></td>
</table>
<h3>Enter password:</h3>
<table>
<td><input name="password" type="PASSWORD"></td>
</table>
<table>
<td><input type="SUBMIT" value="Login"></td>
</table>
```

```
<table>
<td><input type="RESET" value="Cancel"></td>
</table>
</form>
```

Шаблон содержит четыре тега `<INPUT>` — два для ввода имени и пароля, и две обязательные кнопки — **SUBMIT** и **RESET**, аналоги **OK** и **Cancel** в Windows. Кстати, текст на этих кнопках может быть изменен заданием поля `value`.

В итоге страница должна выглядеть так, как показано на рис. 38.3.

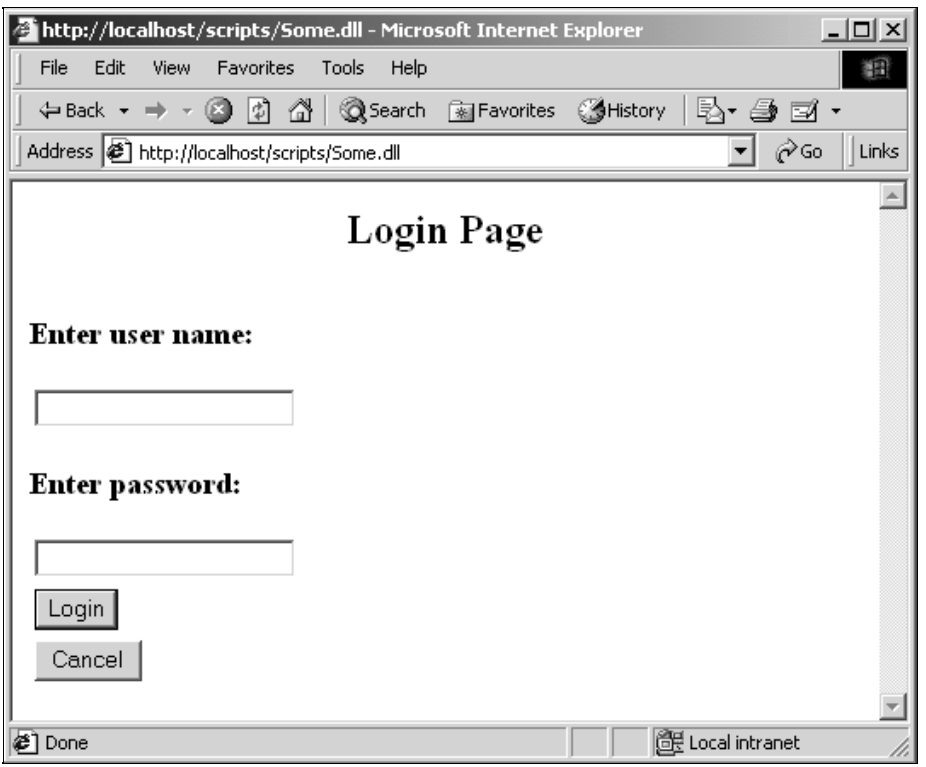


Рис. 38.3. Внешний вид страницы аутентификации пользователей

После того как пользователь введет имя `aaa`, пароль `bbb` и нажмет кнопку **Login**, строка запроса к серверу будет иметь вид:

```
http://localhost/stest/some.dll/validate?username=aaa&password=bbb
```

Примечание

Обратите внимание на то, что пароль передается в незашифрованном виде. Эта система паролей защитит вас от любопытного бездельника, стоящего за спиной, но не от хакера. Для передачи действительно секретной информации нужны другие протоколы, в частности, надстройка над HTTP — Secure HTTP (HTTPS).

Теперь давайте посмотрим, какой же программный код должен обеспечивать проверку данных, введенных пользователем. Метод-обработчик действия `actValidate` для этого запроса может выглядеть так, как представлено в листинге 38.3.

Листинг 38.3. Метод-обработчик действия `actValidate`, обеспечивающий проверку имени и пароля пользователя

```
const Admin: Integer = 0;
      User:  Integer = 1;
      Guest: Integer = 2;
procedure TWebModule1.WebModule1actValidateAction(Sender: TObject;
Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var UserStatus: Integer;
begin
 with ValidateQuery do
 begin
  SQL.Clear;
  SQL.Add('SELECT * FROM Userbase WHERE UserName = :UserName and
  UserPassword = :Password');
  Params[0].AsString := Request.QueryFields.Values['UserName'];
  Params[1].AsString := Request.QueryFields.Values['Password'];
  Open;
  if RecordCount > 0 then
  begin
   UserStatus := FieldByName('UserCategory').AsInteger;
   Response.Content := '<BR><B>Success</B><BR>';
   if UserStatus = Admin
   then
    Response.Content := Response.Content +
                        '<BR>You have Full Access permission<BR>' +
                        '<A HREF="http://' +
                        Request.Host +
                        '/WebPages/AdmMenu.htm"><B>Admin Page</B></A>'
   else
   if UserStatus = User
```

```
   then
    Response.Content := Response.Content +
                        '<BR>You have Modify permission<BR>' +
                        '<A HREF="http://' +
                        Request.Host +
                        '/WebPages/UserMenu.htm"><B>User Page</B></A>'
   else
   if UserStatus = Guest
   then
    Response.Content := Response.Content +
                        '<BR>You have Read permission<BR>' +
                        '<A HREF="http://' +
                        Request.Host +
                        '/WebPages/GuestMenu.htm"><B>Guest Page</B></A>';
  end
  else
   Response.Content := '<BR><B>Wrong UserName or Password</B><BR>';
  Close;
 end;
end;
```

В этом примере сначала при помощи компонента запроса SQL `ValidateQuery` введенные данные проверяются на наличие в базе. Если учетная запись пользователя существует в базе и пароль верен, проверяются его полномочия. В зависимости от категории пользователя ему предоставляется доступ к соответствующей Web-странице. Для этого к ответу приложения добавляется ссылка на специальную страницу. Если имени и пароля в базе данных нет, клиент получает отказ.

Cookies

Серверное Web-приложение может отвечать на запросы анонимных клиентов, но гораздо удобнее, если он "помнит" хотя бы что-нибудь о своих прошлых отношениях с ним (когда и что тот запрашивал), в этом случае прошлые запросы могут стать частью текущего.

Можно, вообще не обременяя клиента, вести на сервере соответствующую базу данных, однако это потребует огромных затрат времени и свободного места. Первыми альтернативный путь придумали разработчики Netscape, которые изобрели так называемые файлы cookies.

Cookies работают следующим образом. Когда приложение идентифицировало клиента, оно может добавить вместе со своим HTTP-ответом специ-

альное дополнительное поле, содержащее идентификатор пользователя и любую нужную серверу информацию, например, время и текст последнего запроса. Если браузер пользователя поддерживает механизм cookies (а это все версии MS Internet Explorer и Netscape Navigator, начиная с третьей), то эта информация будет записана в файл на диске пользователя и в дальнейшем будет присоединяться к каждому запросу этого пользователя к данному серверу.

Для посылки cookies предназначен метод компонента `TWebResponse`

```
procedure SetCookieField(Values: TStrings; const ADomain, APath:
string; AExpires: TDateTime; ASecure: Boolean);
```

Здесь

- `Values` — набор строк в виде `Name=Value`, `ADomain`;
- `APath` — адреса, при обращении по которым следует добавлять cookies;
- `AExpires` — дата истечения срока действия, после которой cookies отсылаться не будут.

Пусть пользователь выбрал какое-нибудь изображение из списка возможных. В дальнейшем мы будем его обрабатывать, поэтому запомним имя изображения и время выбора в cookies:

```
procedure TWebModule1.WebModule1ShowImageAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var CurrentImageName: string;
    MCookies: TStringList;
begin
 CurrentImageName := Request.QueryFields.Values['ImgName'];
 if CurrentImageName<>'' then
  begin
   MCookies := TStringList.Create;
   MCookies.Add('Picture='+CurrentImageName);
   MCookies.Add('Time='+TimeToStr(Now));
   Response.SetCookieField(MCookies, '', '', Date+1, False);
   MCookies.Free;
   Response.Content:='<BR><IMG SRC=http://' +
                     Request.Host +
                     GlobalHTMLPath +
                    '/Images/' +
                     CurrentImageName + '>';
  end
```

```
 else
  Response.Content := 'Image not found';
end;
```

У объекта запроса также есть свойство `Cookies`. Обрабатывая следующий запрос от этого пользователя, мы уже знаем имя выбранного им изображения:

```
CurrentImageName := Request.CookieFields.Values['Picture'];
if CurrentImageName='' then
begin
 Response.Content := 'Image not selected';
 Exit;
end;
else
 Response.Content := 'Image: ' + CurrentImageName;
```

Использование баз данных

Серверные Web-приложения очень часто должны взаимодействовать с базами данных, обеспечивая собственную функциональность или выполняя запросы пользователей. Для реализации этих функций в состав технологии Web Broker включены четыре компонента.

- `TDatSetPageProducer` — создает HTML-текст страницы на основе набора данных и шаблона HTML.
- `TDatSetTableProducer` — генерирует HTML-текст страницы, представляя данные из любого набора данных в табличном виде.
- `TQueryTableProducer` — генерирует HTML-текст страницы, представляя данные из набора данных, созданного параметризованным компонентом запроса SQL только BDE, в табличном виде.
- `TSQLQueryTableProducer` — генерирует HTML-текст страницы, представляя данные из набора данных, созданного параметризованным компонентом запроса SQL только dbExpress, в табличном виде.

Рассмотрим их более детально.

Публикация записей таблиц баз данных

Для того чтобы опубликовать на создаваемой странице не просто информацию, но информацию из базы данных, необходим компонент `TDatasetPageProducer`. Его свойства и методы в большинстве своем совпадают с обычным компонентом-продюсером.

Дополнительно, для связывания с набором данных он имеет свойство

```
property DataSet: TDataSet;
```

через которое можно получить ссылку на любой компонент доступа к данным *(см. часть V)*. Но чтобы создать Web-страницу с данными из соответствующей таблицы базы данных, заполнить только это свойство недостаточно.

Кроме этого, как и стандартный компонент-продюсер, `TDatasetPageProducer` должен иметь шаблон HTML, представляющий собой обычный документ или текст HTML для Web-страницы, но в нем в необходимых местах должны быть включены теги, имена которых совпадут с именами полей этого набора данных в формате `<#FieldName>`. В этом случае компонент автоматически подставит вместо тега значение поля текущей записи набора данных.

Например, для таблицы Country демонстрационной базы данных Delphi простейший шаблон выглядит следующим образом (листинг 38.4).

Листинг 38.4. Пример HTML-шаблона для базы данных

```
<HTML>
<HEAD>
   <TITLE>Biolife Data</TITLE>
</HEAD>
<BODY>
 <TABLE BORDER=0 >
 <TR>
  <TD><B>Name:</B></TD>
  <TD><#Name></TD>
 </TR>
 <TR>
  <TD><B>Capital:</B></TD>
  <TD><#Capital></TD>
 </TR>
 <TR>
  <TD<B>Continent:</B></TD>
  <TD><#Continent></TD>
 </TR>
 <TR>
  <TD<B>Area:</B></TD>
  <TD><#Area></TD>
 </TR>
 <TR>
  <TD<B>Population:</B></TD>
  <TD><#Population></TD>
 </TR>
</TABLE>
</BODY>
</HTML>
```

Однако, как вы уже, наверняка, обратили внимание, для отображения на странице всех записей набора данных с помощью компонента

TDatasetPageProducer придется пойти на ряд ухищрений: создавать сложный шаблон, перемещаться по записям всего набора данных и т. д. Для облегчения этой задачи можно использовать компоненты Web Broker, о которых рассказывается дальше.

Генерация отчетов

При публикации данных в Internet один из самых сложных моментов — генерирование отчетов и выходных форм. Ведь если все таблицы и рамки делать вручную, то можно превратиться из программиста в Web-дизайнера. И тут как нельзя кстати придется пара компонентов: TDatasetTableProducer и TQueryTableProducer. Они и выполняют функцию своеобразного генератора табличных отчетов на Web-страницах.

> **Примечание**
>
> На самом деле компонентов три, но TQueryTableProducer и TSQLQueryTableProducer практически полностью идентичны. Их единственное различие в том, что TSQLQueryTableProducer умеет работать с запросами технологии dbExpress, а TQueryTableProducer — с компонентами запросов BDE. При необходимости использовать запросы других технологий доступа к данным разработчики могут применить компонент TDatasetTableProducer, но настроить параметры запроса вручную.

Первое неотъемлемое свойство любого генератора отчетов для WWW — это компонент-диспетчер, от которого приходят запросы. Второе — это компонент, связывающий генератор с источником данных.

Чем отличаются друг от друга два рассматриваемых компонента? Источником данных и способом его использования.

У TDatasetTableProducer есть свойство

```
property DataSet: TDataSet;
```

которое обеспечивает связь с любым набором данных Delphi.

У TQueryTableProducer имеется свойство

```
property Query: TQuery;
```

которое позволяет получить доступ к набору данных компонента запроса BDE. Однако такая узкая специализация компенсируется замечательной возможностью настраивать параметры запроса TQuery прямо по тексту полученного HTTP-запроса. Предположим, текст свойства SQL в TQuery выглядит следующим образом:

```
SELECT Name, Capital, Area
FROM Countru
WHERE Name = :Name
```

У этого запроса есть параметр `Name`. Если получен такой запрос:

```
http://localhost/test.dll?Name=Canada
```

то его параметр `Name=Canada` будет автоматически передан прямо в компонент запроса, и отчет будет сгенерирован для значения `Canada`.

Напомним, что параметры HTTP-запроса типа `GET` содержатся в свойстве `Request.QueryFields`, а параметры запроса `POST` — в свойстве `Request.ContentFields`. И те, и другие передаются в `TQuery`.

Если вам не нужны параметры, и генерируемый отчет не зависит от содержания запроса, применяйте компонент `TDatasetTableProducer`, т. к. он в большинстве случаев работает быстрее.

Документ, генерируемый при помощи `TDatasetTableProducer` и `TQueryTableProducer`, состоит из четырех частей.

1. Заголовок документа задается свойством:

   ```
   property Caption: string;
   ```

   Значение этого свойства присваивается тегу `<CAPTION>` документа HTML. Выравнивание заголовка определяется как

   ```
   type THTMLCaptionAlignment = (caDefault, caTop, caBottom);
   property CaptionAlignment: THTMLCaptionAlignment;
   ```

   Заголовок можно сформировать динамически. Перед началом его формирования вызывается событие:

   ```
   property OnGetTableCaption: THTMLGetTableCaptionEvent;
   THTMLGetTableCaptionEvent = procedure (Sender: TObject; var Caption:
   string; var Alignment: THTMLCaptionAlignment) of object;
   ```

2. Строки, предшествующие таблице. Чтобы иметь возможность генерировать сложные отчеты, разработчики компонента разрешили предварить таблицу любым HTML-документом:

   ```
   property Header: TStrings;
   ```

3. Собственно таблица. Ее свойства будут описаны далее.

4. Завершающие строки. Они также могут представлять собой любой документ и содержаться в свойстве

   ```
   property Footer: TStrings;
   ```

Чтобы работать с таблицей, имеет смысл задать свойство `DataSet` (`Query`) еще во время разработки. После того как связь с базой данных установлена, нужно заняться редактированием свойства

```
property Columns: THTMLTableColumns;
```

Это свойство и определяет, какие поля базы данных войдут в формируемую таблицу, и как они будут выглядеть. Запустив редактор свойства `Columns` (рис. 38.4) и добавляя новые столбцы, мы можем, во-первых, связать их с полями источника данных, указанного в свойстве `DataSet` (или `Query`).

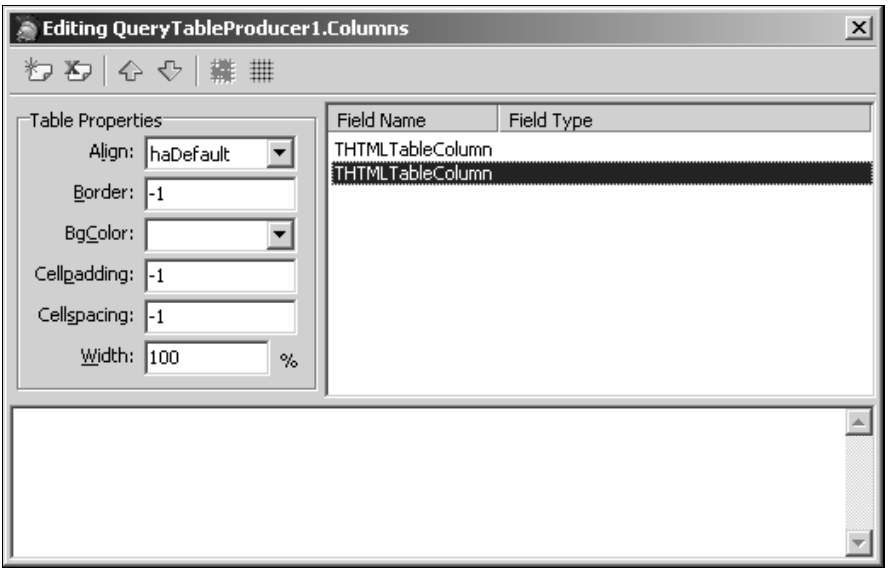


Рис. 38.4. Редактор свойства `Columns` компонентов `TDatasetTableProducer` и `TQueryTableProducer`

Во-вторых, мы имеем возможность задать для них цвет (свойство `BgColor`), горизонтальное и вертикальное выравнивание (`Align` и `vAlign`) и дополнительные теги HTML (свойство `Custom`), а также отдельно цвет, выравнивание и дополнительные теги для заголовка столбца.

Примечание

Для того чтобы нагляднее представить себе значение этих свойств, нужно обратиться к принципам оформления таблиц в языке HTML, а конкретно — к тегам `<TD>` и `<TR>` и возможностям их модификации. Значение перечисленных свойств — `BgColor`, `Align`, `vAlign` — напрямую соответствует тому, что будет содержаться в этих тегах.

В левой части редактора свойства `Columns` можно поменять значения атрибутов всей таблицы, которые входят в состав свойства

```
property TableAttributes: THTMLTableAttributes;
```

Смысл этих свойств поясняет табл. 38.3.

Таблица 38.3. *Свойства объекта* `THTMLTableAttributes`

Свойство	Значение
`property CellSpacing: Integer;`	Расстояние по умолчанию между ячейками таблицы
`property CellPadding: Integer;`	Расстояние по умолчанию от текста в ячейке до ее границ
`property Border: Integer;`	Толщина линии, обрамляющей ячейки таблицы (в пикселах). Значение –1 соответствует отсутствию обрамления
`type THTMLAlign = (haDefault, haLeft, haRight, haCenter);` `property Align: THTMLAlign;`	Способ выравнивания таблицы относительно окна браузера
`property Width: Integer;`	Ширина таблицы в процентах от ширины окна браузера

Для тех, кто знает HTML, можно указать, что значения из `TableAttributes` соответствуют модификаторам тега `<TABLE>`.

Можно отдельно задать и оформление строк, для этого предусмотрено отдельное свойство:

```
property RowAttributes: THTMLTableRowAttributes;
```

Чтобы отчет не получился слишком длинным (а Web-страница — слишком большой и неудобной для загрузки), можно ограничить число строк (т. е. записей из базы данных), которые будут помещены в таблицу:

```
property MaxRows: Integer;
```

Сгенерированный отчет может быть получен вызовом метода `Content` (или его разновидностей — `ContentFromString` и `ContentFromStream`), как и в обычном компоненте-продюсере.

Применение двух описываемых здесь компонентов выглядит на самом деле очень просто:

```
procedure TWebModule1.WebModule1Actions1Action(Sender: TObject; Request:
TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 Response.Content := MyQueryTableProducer.Content;
end;
```

Рассмотрим два оставшихся события, которые возникают в компонентах `TDatasetTableProducer` и `TQueryTableProducer`.

Перед началом генерирования отчета возникает событие:

```
property OnCreateContent: TCreateContentEvent;
type TCreateContentEvent = procedure (Sender: TObject; var Continue:
Boolean) of object;
```

Здесь вы можете проверить состояние базы данных, полномочия клиента и т. д. Если генерировать отчет не следует, установка параметра `Continue` в `False` запретит это.

Важное событие

```
type
THTMLAlign = (haDefault, haLeft, haRight, haCenter);
THTMLVAlign = (haVDefault, haTop, haMiddle, haBottom, haBaseline);

THTMLFormatCellEvent = procedure (Sender: TObject; CellRow, CellColumn:
Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign; var VAlign:
THTMLVAlign; var CustomAttrs, CellData: string) of object;
property OnFormatCell: THTMLFormatCellEvent;
```

возникает при форматировании каждой отдельной ячейки таблицы. Последний параметр — `CellData` — содержит то, что уже было помещено в ячейку самим генератором отчетов. Вы можете модифицировать это содержимое (листинг 38.5).

Листинг 38.5. Пример формирования табличного отчета

```
procedure TWebModule1.MyQTPFormatCell(Sender: TObject; CellRow,
CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
  var s: string;
begin
 if CellRow <> 0 then
  if (CellRow mod 2 = 0)
   then BgColor:='Silver'
   else BgColor:='Gray';
 if (CellColumn = 0) and (CellRow <> 0) then
 begin
  s := CellData;
  CellData := '<A HREF=' +
      Request.ScriptName +
      '/Parts?Ph_Name=' +
```

```
      s +
      ' target="main"> <IMG ALT="[GO!]" SRC="/img/arrow.gif" BORDER=0>' +
      s;
 end;
end;
```

Здесь, во-первых, каждая четная строка окрашивается в один цвет, нечетная — в другой, так таблица выглядит красивее. Во-вторых, сгенерированная `TQueryTableProducer` информация превращается из простого текста в гиперссылку на другую функцию этой библиотеки, дальше она станет ключом для поиска в другой таблице.

Редактирование данных

И в завершение рассмотрим возможности Delphi по редактированию данных. Для этого перейдем к более сложному варианту организации ввода данных, который вытекает из простой задачи: как организовать курсор в HTML-представлении базы данных? Записи уже находятся на экране в виде таблицы, теперь пользователь должен выбрать из них одну или несколько и произвести желаемые действия: удалить, изменить и т. п.

В этом нам опять помогут формы. Предлагаемая последовательность шагов такова.

1. К свойству `Header` компонента `TQueryTableProducer` добавьте HTML-тег `<FORM>` с полями, указывающими на ваше серверное Web-приложение.
2. В процессе генерирования таблицы обрабатывайте событие `OnFormatCell` таким образом, чтобы добавлять в одну из колонок таблицы однострочные редакторы. У каждого элемента управления должен быть уникальный номер, соответствующий номеру строки в таблице (листинг 38.6).

Листинг 38.6. Формирование таблицы с возможностью редактирования данных

```
procedure TWebModule1.StoreQTPFormatCell(Sender: TObject; CellRow,
CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
var s: string;
begin
 if CellRow <> 0 then
  if (CellRow mod 2 = 0)
```

```
    then BgColor:='Silver'
    else BgColor:='Gray';
  if (CellColumn = 0) and (CellRow > 0)
   then CellData := '<INPUT TYPE=HIDDEN NAME=H' +
                    IntToStr(CellRow) +
                    ' VALUE="' +
                    CellData +
                    '">' +
                    CellData;
  if (CellColumn = StoreQTP.Columns.Count - 1) and (CellRow > 0) then
  begin
   CellData := '<INPUT TYPE=CHECKBOX NAME=R' +
               IntToStr(CellRow) +
               ' VALUE=1>Order';
   s := '<BR><INPUT TYPE=TEXT NAME=T' +
       IntToStr(CellRow) +
       ' SIZE=5 MAXLENGTH=8 VALUE="';
   CellData := CellData + s + '">';
  end;
 end;
```

В данном примере таблица форматируется для того, чтобы дать пользователю возможность заказать что-либо из показанной таблицы. Для этого в последний столбец добавляется по одному флажку (с именами `R1...RN`) и по одному текстовому полю (с именами `T1...TN`). Если пользователь что-то выбрал, он должен пометить флажок и ввести число в соседнем поле. В первый столбец вводится невидимое поле `Hidden` (имена `H1...HN`), а его значение будет равно значению поля БД, соответствующего первому столбцу, в текущей записи. Зачем так сделано, будет сказано чуть позже.

3. К свойству `Footer` компонента `TQueryTableProducer` добавляются остальные необходимые теги `<INPUT>`.
4. После того как страница сгенерирована и показана, а пользователь сделал свой выбор, ваша библиотека получит от него запрос. "Соль" примененного приема в том, что из всех созданных флажков `R1...RN` в текст запроса попадут только те, которые помечены пользователем. И если флажок `R1` отмечен, то мы извлекаем из тела запроса значения `H1` и `T1`.

Но зачем нужно поле `H1`, ведь оно не несет никакой дополнительной информации? Дело в том, что пока мы знаем о выбранной записи только ее

номер, а, как вам хорошо известно, в БД выборка по номеру записи не практикуется. Зато у нас есть значение поля БД в элементе `H1` и это поле, взятое из первого столбца, скорее всего окажется ключевым. По нему в любом случае можно отобрать необходимые записи.

Пример простого серверного Web-приложения

И в заключение рассмотрим небольшой пример. Целью его создания является не демонстрация функционирования компонентов Web Broker (достаточно подробно об этом рассказывается ранее), а необходимость обсудить процесс разработки приложения в целом и методы его отладки. Поэтому пример исключительно прост.

При помощи Репозитория Delphi создадим новое серверное Web-приложение. При этом в диалоговом окне (см. рис. 37.2) выберем тип **Web App Debugger executable** — приложение, работающее под управлением утилиты Web App Debugger, входящей в состав Delphi. Назовем приложение Simple Web Server Application.

Примечание

После завершения разработки и отладки мы перенесем готовое приложение на платформу ISAPI.

Главное окно приложения

После создания проекта в среде разработки доступны два модуля. Unit1.pas представляет обычную форму и автоматически будет обеспечивать работу утилиты Web App Debugger. Нас же интересует Unit2.pas, который содержит Web-модуль. Переименуем его в `wmMain`.

Перенесем в Web-модуль из категории **Internet** Палитры инструментов компонент `TPageProducer` и назовем его `MainPage`. Его свойство `HTMLDoc` содержит шаблон главной страницы. На ней доступны две гиперссылки — на две страницы, где отображаются данные из таблицы Country демонстрационной базы данных Delphi в виде списка и таблицы (листинг 38.7).

Листинг 38.7. Шаблон HTML главной страницы приложения SimpleWebApp.exe

```
<BODY>
<H1 ALIGN=center><B>Simple Web Server Application</B></H1>
```

```
<P>
<P>
Here is example of DataSet view using  list style
<A HREF="<#APPNAME>/countrylist">Countries list</A>.
<P>
Here is example of DataSet view using table style
<A HREF="<#APPNAME>/countrytable">Countries table</A>.
<P>
</BODY>
</HTML>
```

Из листинга 38.7 видно, что в шаблоне присутствуют теги `<#APPNAME>`. В методе-обработчике `OnHTMLTag` компонента `MainPage` эти теги заменяются на имя приложения, формируя корректную гиперссылку (листинг 38.8).

Для того чтобы страница `MainPage` открывалась при запуске приложения, в Web-модуле необходимо создать действие (его свойство `Actions`), связать его с компонентом `TPageProducer` (свойство `Producer` действия) и сделать его используемым по умолчанию (свойство `Default` действия). Обратите внимание, что при выборе компонента-продюсера в действии автоматически заполнилось свойство `PathInfo`, которое позволяет идентифицировать действие в запросах. И теперь, если Web-сервер получит запрос

```
http://localhost/SimpleWebApp.exe
```

или

```
http://localhost/SimpleWebApp.exe/MainPage
```

то наше приложение через действие `actMain` свяжет его с компонентом-продюсером `MainPage` и откроет страницу `MainPage`, шаблон которой описан в компоненте.

Для того чтобы гиперссылки на две другие страницы приложения сработали (см. листинг 38.7), необходимо добавить в Web-модуль еще два действия: `actCountryList` и `actCountryTable`. Они должны быть связаны с соответствующими компонентами-продюсерами (см. далее) и их свойства `PathInfo` должны совпадать с запросами для гиперссылок в шаблоне главной страницы.

Просмотр таблицы Country

Теперь займемся таблицей Country. Перенесем из категории **dbGo** в Web-модуль компонент `TADOTable` и настроим его на таблицу Country демонстрационной базы данных Delphi. Для этого выберем файл ..\Program Files\

Common Files\Borland Shared\Data\dbdemos.mdb базы данных Microsoft Access. Подробнее о технологии ADO *см. главу 23*.

Затем из категории **Internet** перенесем в Web-модуль компоненты `TDataSetPageProducer` и `TDataSetTableProducer`. Листинг 38.8 содержит код секции `implementation` Web-модуля приложения SimpleWebApp.

Листинг 38.8. Секция `implementation` Web-модуля приложения SimpleWebApp

```
implementation
uses WebReq, dialogs;
{$R *.DFM}
procedure TwmMain.WebModuleCreate(Sender: TObject);
begin
 SetString(WebAppName, FilePath, GetModuleFileName(hInstance, FilePath,
 SizeOf(FilePath)));
 WebAppName := ExtractFileName(WebAppName);
end;
procedure TwmMain.MainPageHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
 if TagString = 'APPNAME' then
   ReplaceText := WebAppName;
end;
procedure TwmMain.CountryListHTMLTag(Sender: TObject; Tag: TTag; const
TagString: String; TagParams: TStrings; var ReplaceText: String);
var ResponseStr: String;
begin
 if TagString = 'COUNTRYLIST' then
 try
  Country.Open;
  try
   while not Country.EOF do
   begin
    ResponseStr := ResponseStr + Format('<TABLE><TD WIDTH=280>%s</TD>' +
                                        '<TD WIDTH=280>%s</TD>' +
                                        '<TD
WIDTH=280>%s</TD></TABLE>',
     [Country.FieldByName('Name').AsString,
```

```
      Country.FieldByName('Capital').AsString,
      Country.FieldByName('Area').AsString]);
    Country.Next;
   end;
  finally
   Country.Close;
  end;
 except
  on E: Exception do ResponseStr := E.Message;
 end;
 ReplaceText := ResponseStr;
end;
procedure TwmMain.CountryTableFormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
 if CellRow = 0
  then BgColor := 'Gray'
   else
    if CellRow mod 2 = 0
     then BgColor := 'Silver';
end;
initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.WebModuleClass := TwmMain;
end.
```

Первый компонент обеспечит отображение данных из таблицы в виде списка. Для этого необходимо связать его с набором данных — задать в свойстве `DataSet` ссылку на компонент таблицы ADO. Кроме этого, в свойстве `HTMLDoc` мы создадим простейший шаблон, обеспечивающий минимально упорядоченное отображение данных о странах.

При этом шаблон содержит тег `<#COUNTRYLIST>`, который в методе-обработчике `OnHTMLTag` компонента заменяется на данные, полученные из набора данных `Country` (см. листинг 38.8).

Второй компонент обеспечивает представление набора данных `Country` в табличном виде. Набор данных также связывается с компонентом при помощи свойства `DataSet`. И на этом его настройка практически закончена. Небольшие доделки касаются внешнего вида страницы. Например, можно задать

свойства `TableAttributes.Border` и `TableAttributes.CellPadding`. Метод-обработчик `OnFormatCell` обеспечивает заполнение фона нужных строк. В результате во время выполнения компонент `TDataSetTableProducer` обеспечит представление данных, как показано на рис. 38.5.

Примечание

Обратите внимание, что для заполнения обоих страниц используется единственный компонент набора данных.

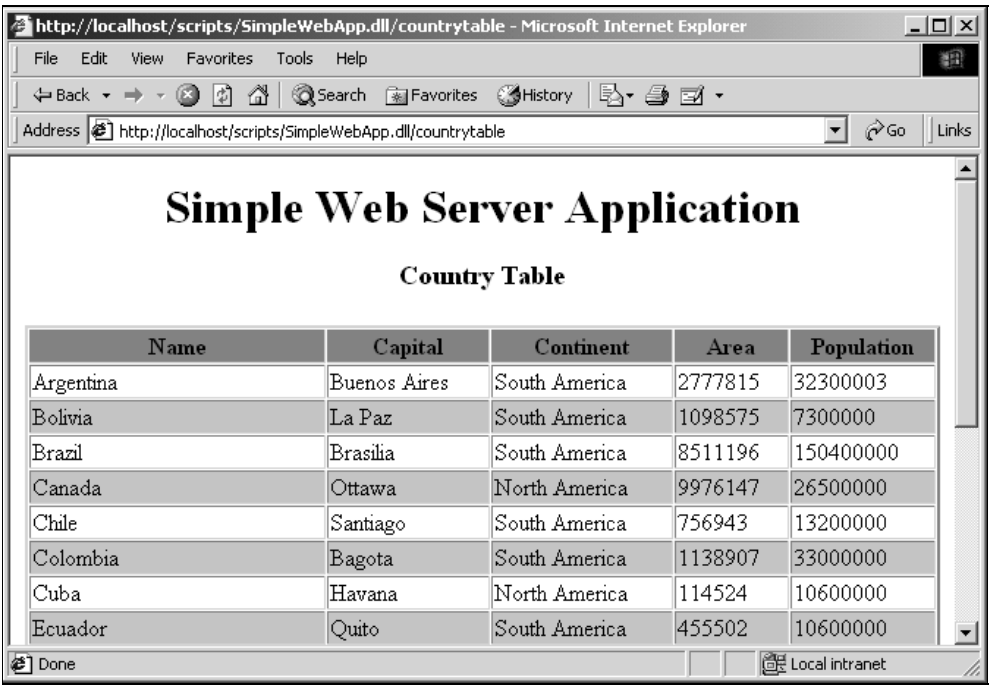


Рис. 38.5. Страница приложения SimpleWebApp.exe

Тестирование приложения

На этом создание приложения закончено. После сохранения и компиляции его можно запустить при помощи утилиты Web App Debugger (рис. 38.6).

Кнопка **Stop** в верхней части окна позволяет обновить список зарегистрированных приложений Web App Debugger. Их список доступен в окне Web-браузера после щелчка на гиперссылке справа от кнопки. Здесь необходимо

выбрать наше приложение и запустить его — в результате появится главное окно `MainPage`.

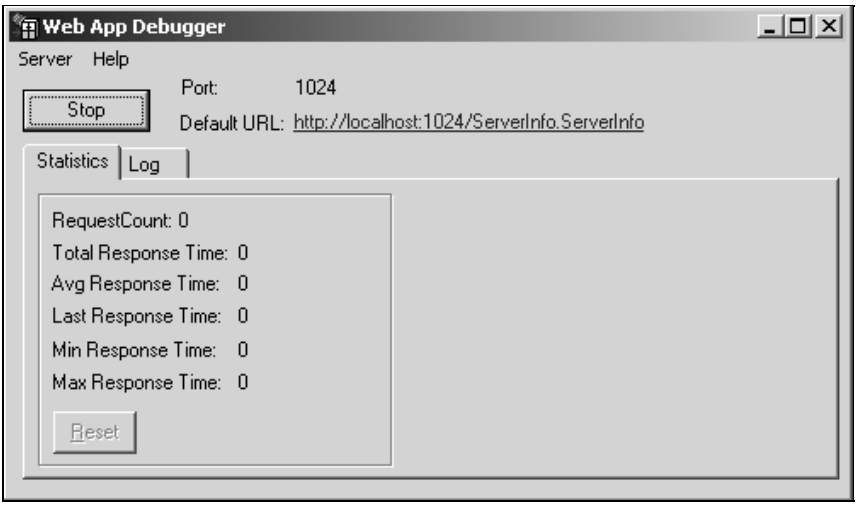


Рис. 38.6. Утилита Web App Debugger позволяет отлаживать серверные Web-приложения без реальных Web-серверов

Резюме

Технология Web Broker позволяет создавать серверные Web-приложения нескольких основных типов для серверов Microsoft Internet Information Server и UNIX Apache. Набор специализированных компонентов и модулей обеспечивает приложению выполнение следующих функций:

- взаимодействие с Web-сервером, обмен запросами и ответами;
- управление выделением памяти и диспетчеризация запросов;
- динамическое создание Web-страниц на основе шаблонов HTML;
- доступ из приложения к базам данных и их использование при создании страниц.

Отличительной особенностью технологии Web Broker является высокая автоматизация операций. Для обеспечения работоспособности большинства компонентов требуется настроить одно-два свойства и применить несколько методов и методов-обработчиков событий.

Дополнительно советуем изучить основополагающий документ RFC 2068 — описание протокола HTTP версии 1.1.

Глава 39

Протокол SOAP и Web-службы. Клиентская часть

По мере развития распределенных приложений остро встала проблема совместимости приложений, работающих на разных платформах. Сейчас средство для вызовов методов удаленных приложений найдено и, главное, одобрено большинством IT-сообщества. Называется оно Simple Object Access Protocol (SOAP). Об этом и пойдет речь в данной главе. Сначала мы убедимся в необходимости SOAP и найдем его место среди других решений. Потом рассмотрим основы SOAP и Web-служб — приложений на его основе. Шаг за шагом пройдем науку создания клиентского и серверного приложения SOAP на Delphi. Вопросам же взаимодействия Web-служб, созданных на Delphi, с решениями других поставщиков, будет полностью посвящена следующая глава.

Почему SOAP?

Софтверные фирмы и независимые разработчики отнюдь не вчера озаботились созданием языка, на котором могли бы разговаривать в Internet Windows и UNIX, Macintosh и мэйнфреймы. Конечно, прочитав предыдущие главы, вы назовете HTTP и ряд других высокоуровневых протоколов. Но что делать, если нужно забраться еще выше? Если нужно не просто запросить гипертекст и показать его пользователю, а организовать взаимодействие двух одновременно работающих на разных платформах приложений?

Есть решения и для этой задачи, например, CORBA. Но у этой действительно кроссплатформенной технологии есть один недостаток — необходимость установки CORBA-клиента у каждого конечного пользователя. Если вы хотите распространять решения на основе VisiBroker — поставляемого с Delphi брокера объектных запросов (Object Request Broker, ORB) — вам следует отдельно позаботиться о приобретении прав на него.

На другом полюсе мы увидим созданное несколько лет назад решение от Microsoft под названием CIS (COM Internet Services). Эта технология —

трюк, позволяющий "удлинить руки" протоколу удаленного вызова объектов DCOM. Как известно, при соединении DCOM-объектов поверх TCP/IP для них выбирается произвольный порт из допустимого диапазона 1024×65 535. CIS — это способ установить соединение и обмениваться данными по порту 80 поверх протокола HTTP.

И вот здесь — ключевой момент наших объяснений. Безопасность в Internet становится все более и более суровой, и тут и там устанавливаются брандмауэры, фильтрующие трафик и действующие по принципу "все, что явно не разрешено, должно быть запрещено". Многие брандмауэры не пропускают вообще ничего, кроме пакетов TCP по портам 80 (вы помните, что этот порт выделен HTTP), 443 (HTTPS) и пакетов UDP по порту 53 (DNS). Причина таких ограничений — многочисленные дыры в разных ОС, которые позволяли хакерам путем сетевых атак по нужному порту как минимум заблокировать сервер, а в худшем случае и запустить на нем "троянского коня".

Вероятно, вы сможете уговорить своего системного администратора и часть клиентов снизить требования к безопасности и открыть часть портов, но трудно предположить, что так же поступят все ваши потенциальные клиенты, если их счет пойдет хотя бы на десятки или сотни. Поэтому на сегодняшний день единственным действительно общепринятым протоколом для установления соединения по Internet следует признать HTTP.

В этом плане CIS позволяет обойти брандмауэры. Но это решение, ориентированное только на DCOM, и ни на что более. О совместимости с другими платформами речь не идет.

Delphi и *TSomeConnection*

Давайте посмотрим с высоты этих глобальных проблем на компоненты Delphi для работы с удаленными базами данных (технология DataSnap, *см. часть VII*). Архитектура DataSnap такова: на серверной стороне есть объект, реализующий интерфейс `IAppServer`, на клиентской — объект `TClientDataSet`, содержащий ссылку на `IAppServer`. В качестве последнего выступает компонент соединения — `TDCOMConnection`, `TSocketConnection`, `TWebConnection` и т. д. Такой компонент также содержит реализацию `IAppServer`, но это не "настоящий" сервер, а средство для маршаллинга запросов от клиента к удаленному приложению. Иными словами, клиент просто обращается к `IAppServer`, не заботясь о том, как данные обмены будут упакованы, переданы, проверены и доставлены.

На серверной стороне маршаллинг реализован по-разному. Объект `TDCOMConnection` не требует серверной части — вызов удаленных интерфейсов осуществляет сама система DCOM. Но поддержка CIS в данном случае

Borland не предусмотрена, так что исключаем `TDCOMConnection` из числа кандидатов на работу в глобальных сетях.

Для `TSocketConnection` предусмотрен специальный Borland Socket Server. Его исполняемый модуль (`scktsrvr.exe`) можно найти в Delphi в папке Bin, а исходные тексты проекта — в Source\Vcl. Обмен данными осуществляется прямо поверх TCP/IP по порту 211 (по умолчанию). Это как раз рассмотренный нами случай — доступ по этому порту, как правило, закрыт.

Следующий компонент имеет многообещающее название — `TWebConnection`. И действительно, связь в нем осуществляется поверх HTTP: на клиентской стороне нужна библиотека WinInet, а на серверной работает ISAPI — библиотека httpsrvr.dll, транслирующая HTTP-запросы в вызовы методов сервера. Но тут проблема не в передаче данных, а в совместимости — ни для чего, кроме соединения `TClientDataSet` с сервером приложений, эта технология не годится. Читатель, знающий DataSnap, возразит: но ведь можно добавить собственные методы и свойства в серверное приложение (точнее, в порождаемый вами интерфейс — потомок `IAppServer`) и экспортировать их на клиентской стороне. Верно, но это решение — частный случай, оно позволяет провести тонкую настройку для клиентов на базе Windows, но не способно ничем помочь клиентам с другими ОС.

SOAP — решение проблем

Итак, резюмируем этот небольшой обзор. Нужен протокол, поддерживаемый всеми основными платформами, представленными в Internet, не требующий сложной клиентской части, и не имеющий проблем с доставкой по Сети и безопасностью.

Такой протокол "постучал в двери" практически сразу за XML. Если есть язык для универсального общения, так давайте писать на нем письма!

Слово "письмо" здесь употреблено не просто как эпитет. Протокол SOAP — это обмен данными в формате XML поверх HTTP, каждая посылка и клиента, и сервера заключается в конверт — отсюда аналогия с письмом.

> ***Важное замечание***
>
> Помните главу о стеке протоколов OSI? Там протокол более высокого уровня лишь опирался на вызовы нижнего уровня, не завися от деталей реализации. Точно так же и SOAP не связан с HTTP неразрывными узами. Пакеты SOAP могут перемещаться с другими протоколами, например, почтовым SMTP — достаточно лишь создать и стандартизовать соответствующую привязку (binding). Но представить себе выражение `y := sin( x )`, вычисляемое по почтовому запросу, трудно. Изначальная спецификация SOAP документирует только привязку к HTTP. Так эти два протокола в сознании многих и стали неразрывными.

Листинг 39.1 иллюстрирует пример посылки SOAP.

Листинг 39.1. Пример SOAP-посылки

```
<?xml version="1.0" ?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <NS1:Login xmlns:NS1="http://tempuri.org/message/">
   <Code xsi:type="xsd:string">User</DCode>
   <Passw xsi:type="xsd:string">secret</DPassw>
  </NS1:Login>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Корневой элемент XML называется `Envelope` — это тот самый "конверт", в который помещается клиентский запрос. В этом запросе (листинг 39.2) некий клиент вызывает метод `Login`, посылая два строковых параметра: `Code` и `Passw`. В ответ он получает значение выходного параметра `Result`, равное 0, — очевидно, признак отсутствия ошибки.

Листинг 39.2. Запрос SOAP

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Body>
 <NS1:LoginResponse xmlns:NS1="http://tempuri.org/message/">
  <Result>0</Result>
 </NS1:LoginResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Конверт, т. е. правила написания писем, — это первая часть стандарта.

Web-службы

Теперь мы можем дать определение Web-службы. Это серверное приложение, доступное через Internet, чей интерфейс опубликован в соответствии с SOAP; язык для такого описания называется WSDL (Web Services Description Language).

Давайте кратко сформулируем возможности Web-служб, не касаясь технических деталей. Вы хотите предоставить в распоряжение клиентов некоторые возможности: данные из вашей информационной системы, машинный перевод с языка на язык, географическую карту, прогноз погоды в нужной местности, партнера для игры в шахматы... Да мало ли что у вас есть полезного! Вы публикуете описание будущей Web-службы на стандартном языке. Это описание говорит клиентам: по такому-то адресу в Internet есть "черный ящик". Если в этот ящик положить такие-то данные в таких-то форматах, то получить от него можно вот такие данные, которые представлены вот в таком формате. Клиенту абсолютно все равно, как реализован сервер и как он обрабатывает данные. На сформулированный по правилам вопрос он должен получить правильный ответ, и не более того.

Некоторые полагают, что SOAP — это чуть ли не новый CIS, этакое развитие технологии DCOM в глобальном масштабе. Это не так, и отличие тут принципиальное.

Создав COM-объект, вы задаете его свойства и вызываете методы. Пока в вашем распоряжении есть ссылка на экземпляр объекта, до тех пор вы можете быть уверены в неизменности свойств, установленных вами. Такие объекты называются объектами, хранящими состояние (stateful).

Как создаются и уничтожаются объекты Web-служб на серверной стороне, клиенту знать не дано. Вызвав, например, метод `SetRate` со значением, равным 0,93, нельзя быть уверенным, что к следующему вызову свойство `Rate` будет иметь прежнее значение. Да, собственно, никто не гарантирует, что вы обратитесь к тому же экземпляру объекта, что и при первом вызове. Объекты SOAP — это пример класса объектов без сохранения состояния (stateless).

Отсюда следует еще одно принципиальное ограничение — невозможность обратных вызовов (callbacks). Для Win32 API и COM характерна передача в вызываемую функцию адреса некоторой своей процедуры, которая исполняется по мере наступления на сервере того или иного события. Поскольку в случае SOAP контекст сервера принципиально неизвестен, об обратных вызовах также следует забыть.

Для удобства разработчика создатели средств для SOAP пытаются имитировать контекст и снабдить его идентификатором сеанса (сессионными файлами Cookie). Некий прогресс на этом пути возможен, но это не должно

вводить вас в заблуждение, SOAP принципиально остается stateless-протоколом. Если ваша задача требует stateful-варианта, вам лучше присмотреться к другим решениям.

Описание протокола WSDL

Протокол WSDL (напоминаем, расшифровывается он как Web Services Description Language) — и есть тот язык, на котором разговаривают клиенты и серверы Web-служб. Текущая его спецификация — 1.1 — находится на стадии утверждения консорциумом W3C, тем не менее все поставщики служб строго ее придерживаются.

Web-служба полностью описывается при помощи следующих базовых понятий.

- **Перечень типов** (types). Основой для обмена данными являются типы, стандартизованные для XML. Но если ваша служба задействует нестандартные (custom) типы, например, структуры данных, их следует описать, указав или приложив соответствующую XML-схему (формата XSD).
- **Сообщение.** Каждое сообщение — это "атом", которыми обмениваются соединяющиеся стороны. Сообщение состоит из одной или более частей (параметров). Для примера: пусть вы вызываете на серверной стороне функцию, чье описание на языке Delphi выглядело бы как `function GetSomeData(Param1: Integer; Param2, Param3: string ↳): double;`
  Входное сообщение для такой функции состояло бы из трех частей — целого числа и двух строковых параметров, а выходное — из одного параметра типа `double`.
- **Тип порта** (Port type) — это описание некоторого набора операций. Операция — это действие над входным сообщением, порождающее выходное сообщение; это максимально похоже на механизм RPC (Remote Procedure Call). Для полноты картины упомянем, что стандарт WSDL определяет и обратный порядок вызова: сервер посылает сообщение, клиент отвечает.
- **Привязка** (Binding) задает связь между типом порта и транспортным протоколом, с помощью которого будут доставляться сообщения.
- **Порт** связывает предыдущие понятия с действительностью. Между типом порта и портом взаимосвязь примерно такая же, как в объектно-ориентированном программировании между описанием класса и объектом, реализующим его. Порт содержит указание на то, что к такому-то типу порта с помощью такой-то привязки возможно обратиться по заданному адресу в Сети.
- Наконец, собственно **Служба** — это поименованный набор портов.

Взаимосвязь этих базовых понятий показана на рис. 39.1.

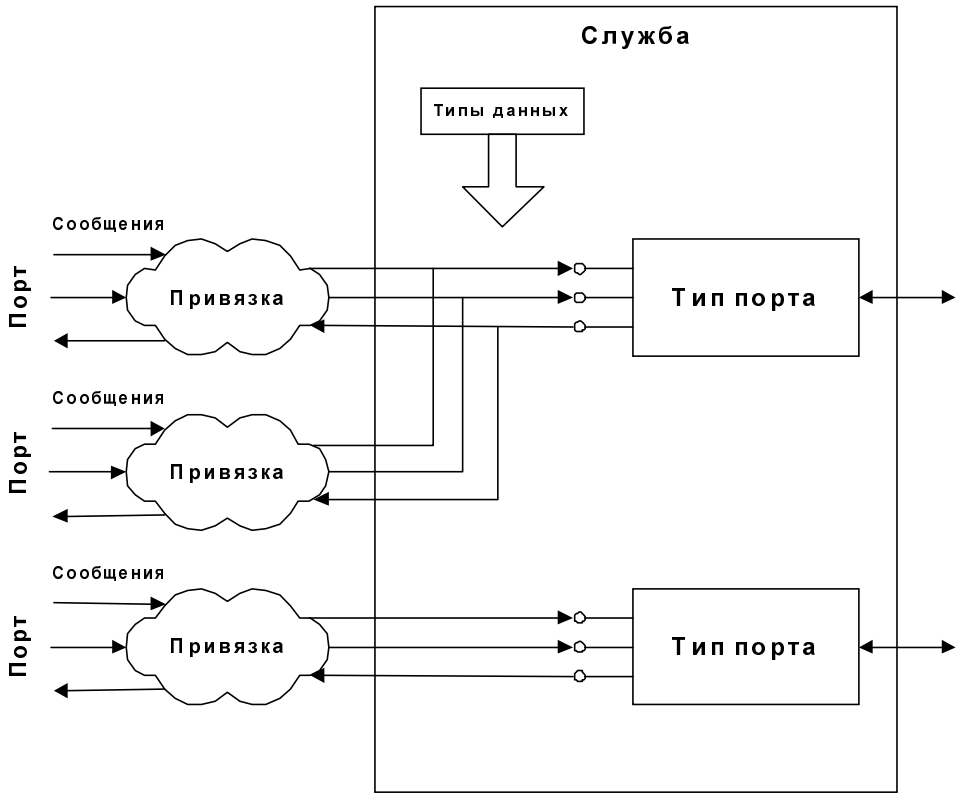


Рис. 39.1. Базовые элементы Web-службы в терминах WSDL

Документы WSDL, что естественно, создаются на XML. Все описанные нами сущности — `<types>`, `<message>`, `<portType>`, `<binding>`, `<port>`, `<service>` — являются зарезервированными именами элементов XML. Окружает все составные части WSDL-документа корневой элемент `<definitions>`, атрибут `name` которого и задает имя службы.

Примечание

Спецификация WSDL допускает элемент `<document>`, предназначенный для "человеческого" документирования содержимого службы и любой из ее составных частей. Здесь разработчики могут оставлять указания и комментарии. Если вам неизвестна или непонятна структура или принципы работы службы, поищите такие элементы в ее WSDL-документе.

Давайте разберем пример WSDL-файла для службы (листинг 39.3), которая способна делать две простые вещи — возвращать в неизменном виде строку

и массив строк, которые были присланы ей для обработки. В дальнейшем мы будем создавать Web-службу на основе этой спецификации.

Листинг 39.3. Пример WSDL-файла службы SimpleEcho

```
<?xml version="1.0"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:xs="http://www.w3.org/2001/XMLSchema" name="ISimpleEchoservice"
targetNamespace="http://tempuri.org/" xmlns:tns="http://tempuri.org/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://www.borland.com/namespaces/Types">
  <types>
    <xs:schema
targetNamespace="http://www.borland.com/namespaces/Types"
xmlns="http://www.borland.com/namespaces/Types">
      <xs:complexType name="TStringDynArray">
        <xs:complexContent>
          <xs:restriction base="soapenc:Array">
            <xs:sequence/>
            <xs:attribute ref="soapenc:arrayType"
n1:arrayType="xs:string[]" xmlns:n1="http://schemas.xmlsoap.org/wsdl/"/>
          </xs:restriction>
        </xs:complexContent>
      </xs:complexType>
    </xs:schema>
  </types>
  <message name="echoStringRequest">
    <part name="Value" type="xs:string"/>
  </message>
  <message name="echoStringResponse">
    <part name="return" type="xs:string"/>
  </message>
  <message name="echoStringArrayRequest">
    <part name="Value" type="ns1:TStringDynArray"/>
  </message>
  <message name="echoStringArrayResponse">
    <part name="return" type="ns1:TStringDynArray"/>
  </message>
  <portType name="ISimpleEcho">
    <operation name="echoString">
```

```
      <input message="tns:echoStringRequest"/>
      <output message="tns:echoStringResponse"/>
    </operation>
    <operation name="echoStringArray">
      <input message="tns:echoStringArrayRequest"/>
      <output message="tns:echoStringArrayResponse"/>
    </operation>
  </portType>
  <binding name="ISimpleEchobinding" type="tns:ISimpleEcho">
    <soap:binding style="rpc" trans-
port="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="echoString">
      <soap:operation soapAction="urn:SimpleEchoIntf-
ISimpleEcho#echoString" style="rpc"/>
      <input>
        <soap:body use="encoded" encoding-
Style="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:SimpleEchoIntf-ISimpleEcho"/>
      </input>
      <output>
        <soap:body use="encoded" encoding-
Style="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:SimpleEchoIntf-ISimpleEcho"/>
      </output>
    </operation>
    <operation name="echoStringArray">
      <soap:operation soapAction="urn:SimpleEchoIntf-
ISimpleEcho#echoStringArray" style="rpc"/>
      <input>
        <soap:body use="encoded" encoding-
Style="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:SimpleEchoIntf-ISimpleEcho"/>
      </input>
      <output>
        <soap:body use="encoded" encoding-
Style="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:SimpleEchoIntf-ISimpleEcho"/>
      </output>
    </operation>
  </binding>
  <service name="ISimpleEchoservice">
    <port name="ISimpleEchoPort" binding="tns:ISimpleEchobinding">
```

```
      <soap:address loca-
tion="http://localhost/scripts/simpleecho.dll/soap/ISimpleEcho"/>
    </port>
  </service>
</definitions>
```

Из чего состоит этот файл?

- Поскольку тип `string` является общеизвестным и фундаментальным, то секция `types` содержит только один тип: он описан схемой элемента `<xs:complexType name="TStringDynArray">`. Кстати, типы динамических массивов `TStringDynArray`, `TIntegerDynArray` и т. п. описаны в Delphi-модуле `Types` и зарегистрированы в модуле `InvokeRegistry`.
- Раздел сообщений содержит четыре элемента — по одному для входного и выходного параметров двух наших функций. Имена их, соответственно, `echoStringRequest`, `echoStringResponse`, `echoStringArrayRequest` и `echoStringArrayResponse`. Теоретически возможно, чтобы одно и то же сообщение использовалось в описании нескольких типов портов; однако на практике стараются этого избегать.
- Единственный тип порта называется `ISimpleEcho` (обратите внимание на это, мы далее будем объяснять, как связаны параметры проекта на Delphi с параметрами службы). Он содержит описания двух операций: `echoString` и `echoStringArray`.
- Привязка указывает на то, что для доставки сообщений будет использоваться протокол SOAP.
- И, наконец, последний элемент WSDL-документа говорит нам о том, что служба `ISimpleEchoService` имеет единственный порт `ISimpleEchoPort`, к которому можно обратиться по адресу **http://localhost/scripts/simpleecho.dll/soap/ISimpleEcho**.

Рассмотрев сам протокол WSDL, давайте разберемся, откуда берутся файлы описания служб и как получить к ним доступ.

Как правило, авторы служб, предназначенных для публичного использования, публикуют его. Загляните на сайт **www.xmethods.com** — один из наиболее посещаемых репозиториев служб. Там каждая служба обязательно сопровождается ссылкой на WSDL-файл.

Службы, созданные на Delphi, автоматически публикуют свое описание. Все они взаимодействуют с Web-сервером посредством стандартов ISAPI/CGI. Библиотека Delphi времени выполнения построена таким образом, что действие по умолчанию (без параметров) и действие ./wsdl выдают "читабельную" информацию для пользователя в табличном виде (рис. 39.2).

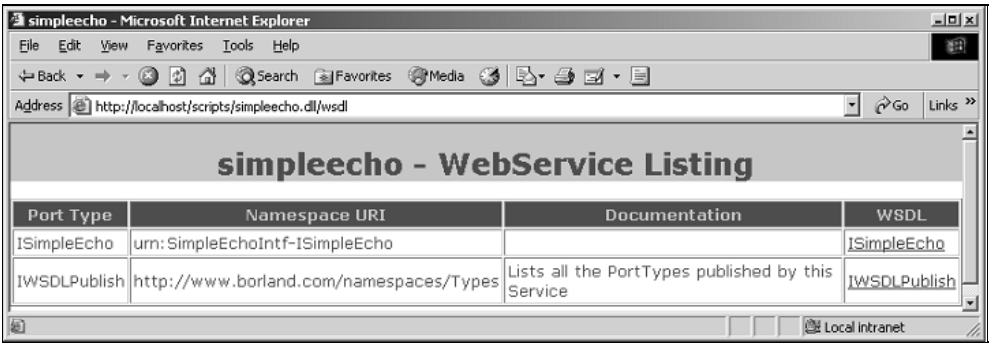


Рис. 39.2. Так Web-служба, созданная средствами Borland, представляет в браузере информацию о своих портах

Если вам нужен только WSDL-файл в формате XML без всякого красивого обрамления, следует задать путь в виде **http://myserver/myservice.dll/wsdl/myPortTypeName**. Помимо портов, заданных разработчиками, среда Delphi автоматически добавляет порт `IWSDLPublish`, который помогает клиентам автоматически получить информацию о службе. Этот тип порта описывается следующим образом:

```
IWSDLPublish = interface(IInvokable)
['{45B6B1C5-A872-C07C-DCC0-798B59768E0E}']
  procedure GetPortTypeList(var PortTypes: TWideStringDynArray); stdcall;
  function  GetWSDLForPortType(const PortType: String): String; stdcall;
  procedure GetTypeSystemsList(const TypeSystems: TWideStringDynArray);
stdcall;
  function  GetXSDForTypeSystem(const TypeSystem: String): String;
stdcall;
end;
```

Зная, что серверная часть службы написана средствами Delphi, вы можете обратиться к ее `IWSDLPublish` и получить список типов портов, их WSDL-файлы и информацию о типах данных.

Если вам нужен полный разбор WSDL-файла до уровня каждого сообщения и его данных, можно, конечно, положиться на XML-анализатор и сделать это вручную. Но гораздо проще обратиться к уже готовым наработкам. Клиент, написанный средствами Delphi, при успешном подключении автоматически получает доступ к интерфейсам `IWSDLDocument` и `IWSDLItems`, которые позволят вам разобраться в описании службы настолько глубоко, насколько необходимо.

Положим, вы получили WSDL-файл необходимой вам службы. Что с ним делать дальше? Как создать клиента, который бы мог обмениваться сообще-

ниями заданных типов? Ответ на этот вопрос зависит от применяемой вами среды разработки. К счастью, для случая Delphi он очень прост. К рассмотрению архитектуры клиентской части мы сейчас и переходим.

Архитектура Web-служб в Delphi

Как и многие другие технологии, SOAP для разработчика на Delphi может предстать в разных обличьях. Если вы хотите создать клиента для удаленных баз данных, вы можете просто взять компонент `TSOAPConnection` и выбрать SOAP как один из возможных протоколов уровня соединения, наряду с HTTP, DCOM или сокетами. В этом случае специальные знания вам почти не понадобятся — вся функциональность протокола SOAP инкапсулирована внутри упомянутого этого компонента.

Если вы пишете службу, не связанную с передачей данных СУБД, или хотите добавить какие-то свои усовершенствования, к вашим услугам мастер создания служб на серверной стороне и компонент `THTTPRIO` — на клиентской.

Наконец, если вам и этого мало, в вашем распоряжении целый ряд компонентов "нижнего" уровня, которые напрямую формируют и принимают посылки протокола SOAP.

Наконец, как клиент, так и сервер могут быть написаны с помощью других инструментов. Следующая глава будет посвящена взаимодействию между Delphi и средствами для создания Web-служб от Microsoft.

Компоненты, которыми вас снабдили разработчики из Borland для работы с Web-службами, находятся в категории **WebServices** Палитры инструментов. Помимо этого, создавая новый проект Web-службы, начните с вызова диалога Репозитория объектов (**File | New | Others**). Там в категории **WebServices** вы найдете заготовки — три для создания серверной части приложения (SOAP Server Application, SOAP Server Data Module, SOAP Server Interface), а также мастер импорта WSDL-файлов для клиента Web-службы.

Клиент Web-службы

Рассмотрение работы Web-служб начнем с клиентской части. Во-первых, возможно, ваша задача сводится только к получению информации от уже существующих служб, причем необязательно созданных при помощи Delphi. Во-вторых, все же клиентская часть проще и поможет нам плавно подойти к написанию серверной части.

Итак, мы начинаем рассмотрение клиента, еще не создав собственного сервера. Однако мы полагаем, что с доступом к Internet у читателя проблем нет: иначе как было бы объяснить его интерес к созданию Web-приложений?

В качестве примера выберем тестовую Web-службу BASE, созданную Borland в рамках проекта SOAPBuilders Interoperability Lab по исследованию совместимости различных реализаций SOAP. Информацию о самом проекте вы найдете по адресам **http://www.whitemesa.com/interop.htm** и **http://www.xmethods.net/ilab**. Местоположение WSDL-файла службы BASE в реализации Borland — **http://soap-server.borland.com/WebServices/Interop/cgi-bin/InteropService.exe/wsdl/InteropTestPortType/**.

Предназначение BASE сколь примитивно, столь и полезно: это эхо, возвращающее назад присланное значение. Причем значение может иметь один из предопределенных 14 типов — от string до массива структур. Отобразить на XML различные типы данных весьма непросто, и "шероховатости" есть если не у всех, то у многих средств для создания Web-служб. На указанных сайтах есть матрицы, показывающие результаты тестирования взаимодействия разных серверных и клиентских реализаций службы BASE — там во многих ячейках стоит унылое FAILED.

Генерация интерфейса Web-службы

Первое, с чего следует начинать создание клиента службы, — с вызова мастера импорта WSDL-файлов в Репозитории объектов (меню **File | New | Other**, категория **WebServices**). Подразумевается, что у вас уже есть готовый открытый проект: обыкновенное приложение, динамическая библиотека или ActiveX.

Роль мастера может быть изложена всего лишь в нескольких словах. Вы указываете ему путь к WSDL-файлу, описывающему Web-службу, после этого он генерирует для вас интерфейс, необходимый для вызовов методов этой службы.

Замечательно то, что вам не нужно думать о технических деталях реализации SOAP в Delphi, вы вообще можете ничего не знать об этом протоколе. Вы получаете объект с методами, в точности соответствующими методам службы. А выглядит это так, как будто вы работаете с самым обычным объектом Автоматизации.

Итак, вызовем мастер и введем в поле **Location of WSDL File or URL** адрес службы BASE, приведенный на рис. 39.3.

Файл описания может находиться как в Сети, так и локально на вашем компьютере.

На втором шаге мастер уже готов показать вам результаты своей работы. Не правда ли, они напоминают результаты импорта библиотеки типов COM (меню **Project | Import type Library**)? Левая панель содержит дерево всех интерфейсов службы вместе с методами, а также специфические типы данных. Справа — модуль кода на языке Delphi, описывающий службу. Нажмите

кнопку **Finish** — и сгенерированный мастером файл добавится в ваш текущий проект (рис. 39.4).

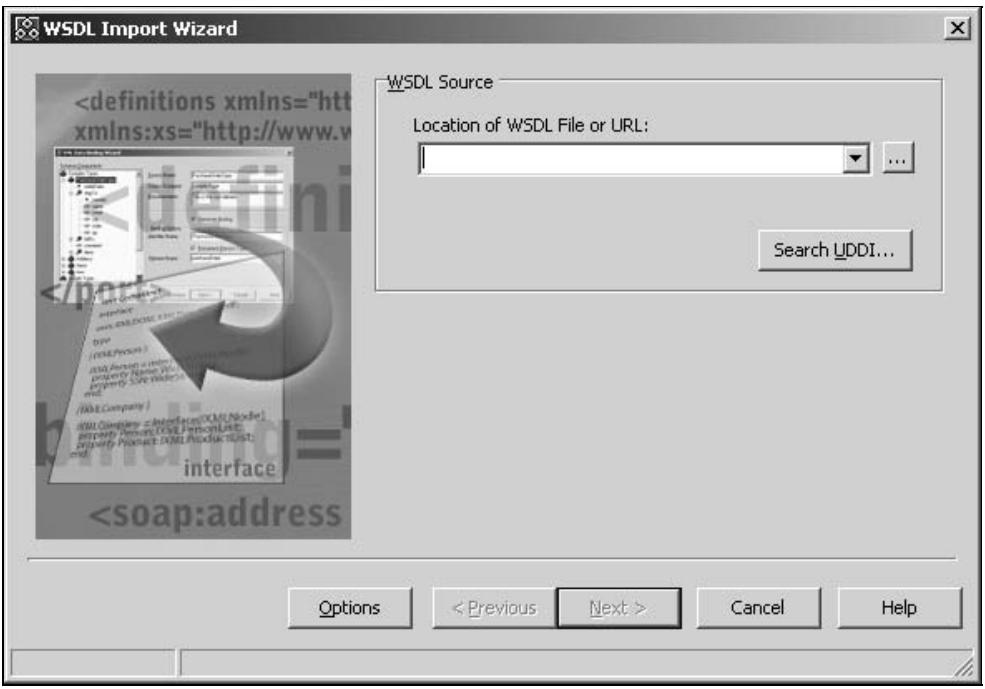


Рис. 39.3. Мастер импорта WSDL-файлов — страница определения адреса службы

Имя сгенерированного PAS-файла будет соответствовать имени службы, оно берется из атрибута `name` элемента `<definitions>` WSDL-документа. Если этот атрибут отсутствует, то имя файла будет совпадать с именем самого документа. В нашем случае мастер создаст InteropTestPortType1.pas.

Примечание

Помимо мастера, который запускается из среды Delphi, вы можете использовать утилиту командной строки wsdlimp.exe, которая находится в папке \BIN. У нее есть общий с мастером файл настроек wsdlimp.ini. Там хранится список уже импортированных описаний служб. Иногда в мастере этот список мешает вводить новые адреса — теперь вы знаете, где его можно отредактировать.

Сохраним файл и проанализируем его содержимое. После описания структур данных следует собственно интересующий нас интерфейс (листинг 39.4).

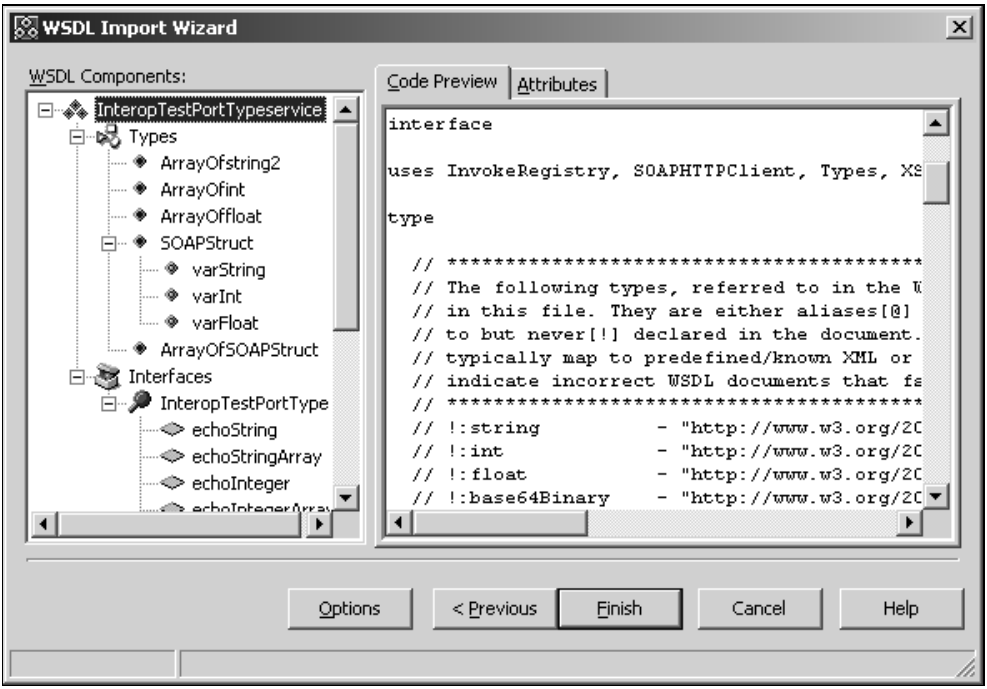


Рис. 39.4. Мастер импорта WSDL-файлов — страница интерфейса службы

Листинг 39.4. Фрагмент WSDL-файла

```
InteropTestPortType = interface(IInvokable)
['{426150D1-7C2C-A74F-954B-DB3ADC067CD8}']
  function  echoString(const inputString: String): String; stdcall;
  function  echoStringArray(const inputStringArray: ArrayOfstring2):
  ArrayOfstring2; stdcall;
  function  echoInteger(const inputInteger: Integer): Integer; stdcall;
  function  echoIntegerArray(const inputIntegerArray: ArrayOfint):
  ArrayOfint; stdcall;
  function  echoFloat(const inputFloat: Single): Single; stdcall;
  function  echoFloatArray(const inputFloatArray: ArrayOffloat):
  ArrayOffloat; stdcall;
  function  echoStruct(const inputStruct: SOAPStruct): SOAPStruct;
  stdcall;
  function  echoStructArray(const inputStructArray: ArrayOfSOAPStruct):
  ArrayOfSOAPStruct; stdcall;
  procedure echoVoid; stdcall;
```

```
    function  echoBase64(const inputBase64: TByteDynArray):
    TByteDynArray; stdcall;
    function  echoDate(const inputDate: TXSDateTime): TXSDateTime;
    stdcall;
    function  echoHexBinary(const inputHexBinary: TXSHexBinary):
    TXSHexBinary; stdcall;
    function  echoDecimal(const inputDecimal: TXSDecimal): TXSDecimal;
    stdcall;
    function  echoBoolean(const inputBoolean: Boolean): Boolean; stdcall;
  end;
```

Четырнадцать методов — это возврат четырнадцати базовых типов данных, определенных рабочей группой SOAP Builders Interoperability Lab для тестирования. Некоторые из них (`Integer`, `Boolean`, `string`) совпадают с оригинальными (native) типами данных Delphi.

Другое подмножество типов описано в стандарте SOAP. Для того чтобы отобразить такие типы на данные Delphi, существуют специальные объекты-переходники. Их имена начинаются с префикса `TXS`. Вам наверняка понадобятся `TXSDateTime`, `TXSDecimal`, `TXSHexBinary` и т. п. Роль переходника — преобразовать данные из SOAP-представления в Delphi и обратно. Каждый такой объект имеет два основных метода: `XSToNative` и `NativeToXS`, а также удобные свойства вроде `AsDate`, `AsByteArray` и т. п.

Наконец, третья группа типов данных — это типы, характерные только для данной службы. Служба BASE определяет пять таких типов, которые вы могли заметить еще в мастере импорта WSDL-файлов. Подробно о типах данных мы будем говорить далее.

Специально подчеркнем связь мастера импорта WSDL-файлов с COM. При всем внешнем сходстве приведенный интерфейс НЕ соответствует реальному COM-объекту. Ни вызовы `CreateOLEObject` с параметром `InteropTestPortType`, ни вызов `CreateCOMObject` с GUID `{426150D1-7C2C-A74F-954B-DB3ADC067CD8}` ничего вам не возвратят. Соответственно, вы можете быть спокойны за содержимое системного реестра: в нем эти интерфейсы не регистрируются. Вы можете импортировать описания десятков служб, единственное, что потом придется удалить, это созданные PAS-файлы.

Но тем не менее COM здесь все же участвует. За описанием интерфейса всегда следует функция, которая носит название `Get<имя службы>`. В нашем случае это

```
function GetInteropTestPortType(UseWSDL: Boolean=System.False; Addr:
string=''): InteropTestPortType;
```

Загляните в ее реализацию. Вы увидите там манипуляции с некоторым объектом `THTTPRIO`. Мы с вами, читатель, как люди заинтересованные, ниже

заглянем в "нутро" этого объекта, который работает "мостом" между Web-службой и COM. А вот функция `Get` как раз создана для тех, кто в это нутро заглядывать не собирается — она предоставляет готовый экземпляр объекта `InteropTestPortType` без "головной боли" о том, откуда он взялся на самом деле. Так что простейший сеанс общения с Web-службой BASE выглядит следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var Port : InteropTestPortType;
begin
 try
  Port := GetInteropTestPortType;
  ShowMessage( Port.echoString('Hello, world!') );
 finally
  Port := nil;
 end;
end;
```

Переменная `Port` получает "настоящий" экземпляр COM-объекта, и поэтому после использования ей следует обязательно присвоить значение `nil`.

Итак, добавьте на пустую форму вашего проекта кнопку `Button1` и свяжите ее событие `OnClick` с этим методом — получите первый "привет" из мира Web-служб.

В примере мы вызвали функцию `GetInteropTestPortType` без параметров, а между тем их у нее два, хотя и необязательных. Они нужны для того, чтобы создавать объект не по тому WSDL-файлу или URL, который использовался мастером импорта. Допустим, у вас на диске хранится копия WSDL-документа, необходимая для отладочных целей (об этом далее). Тогда параметр `UseWSDL` должен быть равен `True`, а `Addr` — указывать на отладочный файл:

```
Port := GetInteropTestPortType( True,
'c:\MyFolder\InteropTestPortType.wsdl');
```

Вот на этом первый шаг в мир Web-служб можно считать сделанным. Ваших знаний теперь достаточно, чтобы найти описание интересующей службы, создать для нее клиента и вызвать методы. Но вряд ли вас удовлетворят только самые простые случаи — поэтому идем дальше.

O, RIO, RIO

Абрревиатура RIO — ключевая для понимания клиентского интерфейса SOAP в Delphi — не имеет отношения к Рио-де-Жанейро. RIO — это Remote Interfaced Object. Так называют объект, умеющий "притворяться" Web-

службой. Как это возможно? Все мы привыкли к статическим описаниям интерфейсов COM и классов Object Pascal. Компилятор читает описание, которое, как правило, публикуется на языке IDL, и генерирует вызовы Таблицы виртуальных методов (ТВМ) интерфейса. Если описание соответствовало действительности, то вызовы методов будут успешными независимо от языка реализации COM-объекта и вызвавшего его приложения.

Но ведь мы не всегда знаем описание интерфейса Web-службы заранее! Соответственно, мы не можем поручить эту работу компилятору.

И вот тут напрашивается дерзкая идея: а почему бы и саму Таблицу виртуальных методов не формировать динамически, а не статически? Компилятор знает структуру ТВМ, потому что он обработал PAS-файл описания интерфейсов службы, сгенерированный вышеупомянутым мастером. А откуда в ней возьмутся методы и как они будут вызываться, ему все равно.

В Borland сумели реализовать такую идею. Так на нашей сцене появляется `TRIO`. Первоначально этот объект Delphi не поддерживает никаких интерфейсов, кроме `IUnknown`. Но когда вы обратитесь к нему за другим интерфейсом с помощью метода `QueryInterface` или в более привычном синтаксисе для программистов Delphi

```
Obj := RIO as InteropTestPortType;
```

то `RIO` попытается выполнить следующие действия.

1. Считать описание затребованного интерфейса из специального хранилища `InvokeRegistry`. В нем хранятся описания всех интерфейсов Web-служб; подробнее о нем речь пойдет далее.
2. По полученному описанию сгенерировать ТВМ и методы-переходники (Stubs), которые предназначены для приведения в соответствие правил соглашения о вызове (calling convention) и типов параметров данного метода.
3. Возвратить ссылку на созданную ТВМ и увеличить *свой* счетчик обращений на единицу — отныне `RIO` работает как экземпляр объекта, поддерживающего запрошенный интерфейс.

Сам класс `TRIO` — не более чем общая схема реализации идеи удаленного вызова. На Палитре компонентов Delphi вы найдете компонент `THTTPRIO`, что означает адаптацию `RIO` к передаче вызовов методов в формате протокола SOAP поверх транспортного протокола HTTP. Напомним, что возможны другие варианты кодирования и передачи сообщений SOAP. В этом случае появятся и другие потомки `TRIO`. Пока же представитель этого семейства только один.

Создавать экземпляр этого необычного объекта можно двумя способами.

- Как обычный компонент. Поместите на форму экземпляр `THTTPRIO` из Палитры и задайте его свойства в режиме разработки. В этом случае хо-

зяином (owner) компонента будет форма, фрейм или другой компонент. Он же будет отвечать за освобождение экземпляра `THTTPRIO`.

- Динамически, вызвав конструктор `Create` с параметром `Owner=nil`. Именно так делается по умолчанию внутри функции `GetInteropTestPortType` и всех аналогичных функций, создаваемых мастером импорта WSDL-файлов. В этом случае полную ответственность за жизненный цикл объекта несет его создатель.

Напомним, что освобождение экземпляров объектов является краеугольным камнем COM. При каждом затребовании экземпляра вызывается метод `_addRef`, увеличивающий счетчик обращений на единицу. Обнуляя ссылку (присваивая ей `nil`), вы тем самым отказываетесь от экземпляра и говорите библиотеке времени выполнения (RTL) Delphi вызвать метод `_Release` и уменьшить число обращений. Когда он становится равным нулю, экземпляр объекта должен уничтожиться.

Но код компонента `TRIO` и его потомки написаны таким образом, что делает он это *только* в том случае, если он не имеет хозяина-потомка `TComponent`. Иными словами, если вы создали `RIO` как компонент, пользуйтесь им как обычным компонентом, не задумываясь об экземплярах. Если вам больше привычна парадигма COM, то пользуйтесь им как обычным объектом COM.

Специально для тех, кто запутался в этих правилах, обычно скрытое свойство `RefCount` вынесено в секцию `public`, где оно доступно по чтению:

```
property RefCount: Integer;
```

Если вы не уверены в своем коде, контролируйте значение `RefCount`: перед вызовом `TRIO.Free` значение числа ссылок должно быть нулевым.

Как связаться с нужной службой

Теперь пришла очередь свойств, которые указывают на службу, к которой мы подключаем наш объект `THTTPRIO`. Здесь тоже возможны два подхода.

Первый подход в точности совпадает с логикой работы мастера импорта.

Три свойства, в указанном порядке:

```
property WSDLLocation: string;
property Service: string;
property Port: string;
```

доведут вас до конкретного интерфейса. Сначала задайте местоположение WSDL, при этом выпадающий список покажет вам адреса уже знакомых служб. Напомним, что адрес может задаваться не только URL, но и путем к локальному файлу на диске.

Когда файл будет указан и успешно пройдет синтаксический разбор, свойства `Service` и `Port` пополнятся коллекциями доступных по этому адресу служб и их портов.

Второй путь приемлем, как правило, если и клиент, и сервер написаны на Delphi или Kylix. Когда вы знаете о службе все, то зачем делать промежуточный шаг и считывать WSDL? Обращайтесь напрямую к порту, чей адрес задается свойством URL. Напоминаем, что этот адрес должен быть прописан в элементе `<port>` WSDL-файла:

```
<service name="InteropTestPortTypeservice">
  <port name="InteropTestPortTypePort"
  binding="tns:InteropTestPortTypebinding">
   <soap:address location="http://
   soap-server.borland.com/WebServices/Interop/
   cgi-bin/InteropService.exe/soap/InteropTestPortType" />
  </port>
</service>
```

Для служб, созданных средствами Borland, этот адрес похож на адрес WSDL-файла, с той разницей, что действие `/wsdl` заменяется на действие `/soap` (как в данном примере).

Естественно, свойства URL и `WSDLLocation/Service/Port` являются взаимоисключающими — при настройке одного значение другого (других) очищается.

Проиллюстрируем эти свойства еще одним простым примером. На форму нашего первого клиента службы BASE (вы еще не закрыли ее?) переместите из Палитры компонентов `THTTPRIO` и настройте его свойства `WSDLLocation/Service/Port` на `http://soap-server.borland.com/WebServices/Interop/cgi-bin/InteropService.exe/wsdl/InteropTestPortType/`, `InteropTestPortTypeservice` и `InteropTestPortTypePort` соответственно.

Добавьте еще одну кнопку и на ее `OnClick` следующий код:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
   ShowMessage( (HTTPRIO1 as InteropTestPortType).echoString
  ('Hello, world!') );
end;
```

При этом заботиться о счетчике обращений не обязательно — код очистки RTL Delphi, выполняемый по выходу из метода `Button2Click`, приведет его в порядок.

У объекта `THTTPRIO` есть всего два события `OnBeforeExecute` и `OnAfterExecute`:

```
property OnBeforeExecute: TBeforeExecuteEvent;
TBeforeExecuteEvent = procedure(const MethodName: string;
var SOAPRequest: InvString) of object;
```

```
property OnAfterExecute: TAfterExecuteEvent;

TAfterExecuteEvent  = procedure(const MethodName: string; SOAPResponse:
TStream) of object;
```

Они возникают непосредственно перед и после исполнения метода службы. В частности, эти события могут быть полезны для анализа получаемой и отправляемой информации — параметры `SOAPRequest` и `SOAPResponse` есть ни что иное, как посылки одноименного протокола.

Решение коммуникационных проблем

Надежность и скорость работы служб зависит, прежде всего, от транспорта, доставляющего сообщения. В данном случае это HTTP. Для Delphi 6 HTTP реализован путем вызова библиотеки WinInet.dll, поставляемой вместе с Windows и Internet Explorer и содержащей весь механизм (engine) для работы с высокоуровневыми протоколами, встроенными в IE: HTTP, HTTPS, FTP и Gopher. Поскольку Windows без Internet представить невозможно уже последние лет шесть, такой подход кажется оправданным. Тем более что WinInet является достаточно надежной и отработанной реализацией перечисленных протоколов.

С другой стороны, независимость от типа платформы и, в частности, независимость от Microsoft также может только приветствоваться. Для Kylix избрана реализация HTTP, поставляемая с библиотеками Indy (о них мы рассказывали в предыдущих главах). Если вы посмотрите на код модуля SoapHTTPTrans.PAS (папка ..\Source\Win32\Soap), то увидите там много директив условной компиляции: `{$IFDEF USE_INDY}`. Имея определенные навыки, можно перекомпилировать модуль и получить реализацию HTTP на базе Indy и в Delphi. Более того, в конференциях по Delphi, в частности, в **borland.public.delphi.webservices.soap**, можно найти и привязки к другим реализациям HTTP.

Это стало возможным благодаря модульной конструкции `RIO`. Как ни странно, кратко пройдясь по собственным свойствам компонента `THTTPRIO`, мы практически их уже исчерпали. Зато у него есть два дочерних компонента со своими (важными!) свойствами — в Инспекторе объектов вы видите их как `Converter` и `HTTPWebNode`.

Если вы ищете объяснение всем шести компонентам на странице **WebServices** Палитры компонентов Delphi, то вот вам еще два, которые вынесены туда: `TOPToSoapDomConvert` и `THTTPReqResp`. Это, как вы, наверное, догадались, и есть `Converter` и `HTTPWebNode`.

Первый компонент — `Converter` — отвечает за преобразование вызовов между протоколом SOAP и приложением. Все важное, что у него опубликовано, — это свойство `Options`. Его настройки в основном влияют на совместимость с другими клиентами и серверами Web-служб. Поскольку в Delphi 6

они общие и для серверов служб, мы рассмотрим их подробнее в следующей главе.

Второй — `THTTPReqResp` — отвечает за пересылку данных. Его название собственно и произошло от терминов HTTP, Request и Response.

Все это так, но зачем он нужен лично вам? Ответ: для параметризации настроек соединения. Задачи, которые можно решить с помощью настроек, таковы.

1. Определение настроек proxy-сервера. Свойство

   ```
   property Proxy: string
   ```

   указывает на адрес такого сервера, через который будет происходить соединение. Внимание: отсутствие значения у этого свойства *не означает*, что соединение будет происходить без прокси. В этом случае выбираются стандартные настройки библиотеки WinInet. Иными словами, вы будете пользоваться теми же настройками, что и ваш Internet Explorer.

   Если вы когда-либо смотрели, как настраивается proxy в Internet Explorer, вы не могли пройти мимо настройки, которая называется так же, как и свойство

   ```
   property ProxyByPass: string;
   ```

   Если с некоторыми из серверов надо соединяться, минуя proxy, то их список можно задать в этом свойстве через точку с запятой.

   Наконец, свойство

   ```
   property Agent: string;
   ```

   задает значение одноименного заголовка протокола HTTP. Некоторые брандмауэры (firewalls) фильтруют HTTP-трафик по значению этого поля, так что позаботьтесь об агенте 'Borland SOAP 1.1' (и 'DataSnap', если вы пользуетесь компонентом `TWebConnection`).

2. Авторизация пользователя службы. Если сервер службы или proxy-сервер требует авторизации, в свойствах

   ```
   property UserName: string;
   property Password: string;
   ```

   можно задать необходимые имя пользователя и пароль. Для многих клиентских приложений системное диалоговое окно ввода пароля просто недопустимо. Задав значения этих свойств, вы можете этого избежать.

Примечание

Компонент `THTTPRIO` ведет себя по-разному, в зависимости от того, помещен ли он на форму в режиме разработки (design mode) или создан динамически. В первом случае отсутствие Username и Password приведет к возврату серве-

ром кода ошибки 401 и генерации соответствующей исключительной ситуации. Динамически созданный `THTTPRIO`, очевидно, не инициализирует значений этих свойств: вы увидите системный диалог авторизации, выглядящий в зависимости от типа Web-сервера и принятого способа авторизации.

Мы достаточно времени посвятили сертификатам, чтобы сейчас обойти вопрос защищенного соединения. Если вы собираетесь соединяться с Web-сервером по SSL, то простого изменения префикса протокола с HTTP:// на HTTPS:// будет вполне достаточно, т. к. библиотека WinInet должна правильно распознать настройку.

Примечание

Если вы же пишете серверную часть службы, синхронно поменяйте название протокола в WSDL-файле — не сделать этого типичная ошибка.

В свойстве `InvokeOptions` есть флаг под названием `soIgnoreInvalidCerts`. Он предназначен для случаев, когда сервер имеет просроченный или недействительный по другим причинам сертификат. Установка этого флага позволяет игнорировать такие сертификаты без остановки на диалог и информирования пользователя.

Если вы вдобавок используете для авторизации клиентские сертификаты, то тут ситуация сложнее. Код класса `THTTPReqResp` не содержит способов выбора нужного сертификата из нескольких возможных. К тому же, разные реализации WinInet ведут себя с клиентскими сертификатами по-разному: некоторые не поддерживают эту возможность или содержат ошибки, другие автоматически выбирают первый доступный из клиентских сертификатов, некоторые открывают для этого диалоговое окно выбора. Все это следует иметь в виду; если все же решились защитить службу с помощью такого метода авторизации, вам, очевидно, предстоит частично переписать код модуля `SOAPHTTPTrans`. Функция из WinInet под названием `InternetSetOption` позволяет выбрать контекст нужного сертификата.

Примечание

То же самое можно сказать и о еще одном важном параметре — величине тайм-аута соединения на клиентской стороне. Пока его регулировать невозможно.

Компонент `THTTPReqResp` имеет ряд характерных для HTTP методов — `Connect`, `Send`, `Receive` и др. Но рассматривать их здесь излишне, т. к., во-первых, они не являются виртуальными и перекрыть их нельзя и, во-вторых, поскольку они вызываются еще до первого вызова методов службы.

Что такое *InvokeRegistry*. Регистрация интерфейсов и типов данных

Если вы внимательно смотрели на описание нашего первого интерфейса, то заметили, что `InteropTestPortType` порожден от `IInvokable`. Что это за интерфейс и чем он отличается от прародителя всех интерфейсов `IUnknown` (он же — `IInterface` в Delphi)? Смотрим исходный текст модуля `System`:

```
{$M+}
  IInvokable = interface(IInterface)
  end;
{$M-}
```

Оказывается все дело в директиве `{$M+}`, она же — `{$TYPEINFO ON}`. Эта директива компилятора включает генерацию Информации о типе (Runtime Type Information, RTTI).

RTTI — это механизм, на котором держится визуальная среда программирования Delphi. Поля и методы классов, объявленные как published, а также другие свойства (имя класса, размер экземпляра и т. п.) попадают в специальную таблицу, указатель на которую есть у каждого экземпляра данного класса. Помните методы класса `TObject`, такие как `ClassName`, `ClassInfo`, `MethodAddress`, `MethodName` и т. п.? Все это суть есть навигация по RTTI.

Когда вы сохраняете форму из состава своего проекта, она сама и расположенные на ней компоненты должны быть записаны в файл с расширением dfm. При повторном открытии значения опубликованных свойств сохраняются. Почему? Потому что при записи происходит извлечение информации из RTTI и перекодирование ее в текстовый или бинарный формат (в зависимости от того, как у вас хранятся формы). Свойства, не являющиеся опубликованными, в принципе не могут быть сохранены.

Почему же многие в первый раз слышат о директиве `{$M}`? Потому что она работает таким образом, что если родительский класс был скомпилирован в состоянии `{$M+}`, то все его потомки будут иметь RTTI, даже если опубликованные методы и свойства появились непосредственно в классе-потомке. "Точкой отсчета" для классов является `TPersistent` — а, следовательно, и все компоненты VCL — его потомки.

Теперь пару слов об интерфейсах. С классами, надеемся, все более или менее понятно — вот описание класса, вот экземпляр, вот RTTI. Но что означает директива `{$M+}` перед описанием интерфейса? А то, что данный интерфейс и все классы Delphi, реализующие его, будут включать информацию о методах интерфейса в RTTI, как если бы они были опубликованными методами самого класса.

Такое новшество впервые возникло в Delphi 6, и появилось оно специально для создания Web-служб и их клиентов. В самом деле, сходство задач оче-

видно: при считывании файла формы среда Delphi воссоздает объект по его текстовому описанию, то же самое должен сделать клиент службы, но только не из DFM-, а из WSDL-файла.

Но одной RTTI мало — нужен специальный репозиторий, где бы хранилась информация обо всех интерфейсах, классах и типах данных, которые принимают участие в работе Web-служб. Таких репозиториев два. Один — глобальный объект, доступный как `InvRegistry` — хранит описания интерфейсов (регистрация в нем обязательна и для клиентской, и для серверной частей) и классов, реализующих эти интерфейсы (только для серверной части). Если вы генерируете модули интерфейса при помощи мастера, то регистрация будет осуществлена за вас — посмотрите на секцию инициализации сгенерированного для примера модуля InteropTestPortType1.pas.

Второй объект — `RemClassRegistry` — хранит описания зарегистрированных типов данных. Неверным было бы считать, что стандартные типы данных Delphi регистрировать не надо. Просто они уже зарегистрированы в модуле `InvokeRegistry`. Это `Boolean`, `Char`, `Byte`, `SmallInt`, `Word`, `Integer`, `Cardinal`, `Int64`, `Single`, `Double`, `Extended`, `String`, `WideString`, `Currency`, `Variant`. Зарегистрированы также динамические массивы этих типов (кроме `Char`, `Currency` и `Variant`).

Нестандартные типы и структуры должны быть зарегистрированы в вашем приложении. Опять же, мастер импорта WSDL делает это за вас. В нашем примере со службой BASE регистрации подлежат структура `SOAPStruct` и массив таких структур.

И тут нас поджидает небольшое неудобство. Если массивы регистрируются естественным путем, то для регистрации структур надо создавать описывающие их классы. Точно так же приходится поступать с типами, которые не могут быть отображены на строки XML известным RTL Delphi способом.

Добавим к нашему примеру еще одну кнопку и несколько строк кода:

```
procedure TForm1.Button3Click(Sender: TObject);
var ssTest : SOAPStruct;
begin
 Randomize;
 ssTest := SOAPStruct.Create;
 try
  ssTest.varString := 'Hello, world!';
  ssTest.varInt := Length( ssTest.varString );
  ssTest.varFloat := Random;
  ssTest := (HTTPRIO1 as InteropTestPortType).echoStruct( sstest );
  ShowMessage( ssTest.varString );
```

```
 finally
  ssTest.Free;
 end;
end;
```

Приходится помнить, что нестандартные типы — это классы, и обращаться с ними соответствующим образом. После создания и использования объекта `ssTest` экземпляр подлежит освобождению.

Есть несколько очень часто употребляемых типов данных, которые для работы в службах должны иметь описывающие их классы. Это дата и время, а также массив двоичных данных. Разработчики Delphi снабдили нас такими классами — они находятся в модуле `XSBuiltIns` и называются, соответственно, `TXSTime`, `TXSDate TXSDateTime` и `TXSHexBinary`. Буквы XS в названии означают XML Schema.

Все XS-классы объединяет то, что они имеют виртуальные методы

```
procedure XSToNative(Value: WideString);
function  NativeToXS: WideString;
```

Это собственно преобразователи представления данных из внутреннего формата в строковый. Но вам редко придется вызывать эти методы. Гораздо удобнее вызывать методы типа `AsSomething`:

```
property AsDateTime: TDateTime;
```

для даты/времени,

```
property  AsByteArray: TByteDynArray
```

для двоичного массива и т. п.

Сгенерированные из WSDL-интерфейсы служб часто содержат в качестве параметров именно XS-классы. Тогда работа с ними почти не будет отличаться от того, что мы проделали с `SOAPStruct`.

Вот алгоритм действий.

1. Создайте экземпляр класса.
2. При помощи `AsSomething` присвойте ему нужное значение.
3. Передайте его в качестве параметра. За преобразование он отвечает сам.
4. Если служба возвращает значение этого типа, прочитайте полученный результат при помощи того же метода.

Если ваша задача не может быть решена при помощи мастера и описанных типов, то класс, представляющий ваш тип данных, придется писать самому. Первый совет, который нужно дать здесь, — не делать этого. Мы уже не раз говорили о проблеме совместимости реализаций SOAP. Не факт, что другие средства разработки смогут правильно представить и об-

работать ваш тип данных. Поэтому, по возможности, следует ограничиться стандартными типами.

Если, тем не менее, это невозможно, следует породить дочерний класс от `TRemotable` (для массивов и структур) или от `TRemotableXS` (для данных, нуждающихся в преобразовании в строчный тип). Регистрируются эти типы путем вызова `RemClassRegistry.RegisterXSClass` (для класса) или `RemClassRegistry.RegisterXSInfo` (для прочих типов).

Резюме

В этой главе мы научились создавать простейшее клиентское приложение, использующее протокол SOAP. Сила Web-служб в их самодокументируемости. Вооружившись одним-единственным компонентом `THTTPRIO`, вы можете "пожирать" WSDL-файлы и перепробовать в работе любые заинтересовавшие вас службы, разумеется, те, которые предоставляют услуги бесплатно. Для многих предоставление полезных служб уже стало бизнесом. Для начала поисков можно порекомендовать сайт **http://www.xmethods.net**.

Ну, а мы переходим к следующему шагу разработки — серверу Web-службы.

Глава 40

Серверная часть Web-службы

Из трех серверных заготовок в Репозитории объектов *(см. главу 39)* главной является первая — SOAP Server Application. Две прочие — SOAP Server Data Module и SOAP Server Interface — составные части первой.

Итак, первый вопрос, на который предстоит ответить, выбрав SOAP Server Application, — тип серверного приложения (см. рис. 37.2). Выбор из трех альтернатив на первый взгляд сложен. Но на самом деле все просто, давайте лишь разберемся. Если вы работаете с Microsoft IIS, то первая альтернатива — ISAPI DLL — предпочтительный вид приложения для него. Приложения CGI и WinCGI запускаются как исполняемые модули в отдельных процессах, что сильно повышает накладные расходы. CGI выбирают, как правило, в тех случаях, когда остальные альтернативы по тем или иным причинам недоступны или нежелательны.

С другой стороны, если вам предстоит длительная отладка, предпочтения могут измениться. Сервер IIS кэширует модули ISAPI. Если вы изменяете свой модуль, вам предстоит каждый раз перезагружать IIS. В принципе, можно отключить кэширование, но у всех ли есть такая возможность? Приложения CGI лишены этого недостатка.

Наконец, если вы хотите отлаживать серверную часть в среде Delphi, Borland включил в Delphi специальную утилиту — WebApp Debugger.

Мастер создает главный Web-модуль приложения (форму класса `TWebModule`) и далее предлагает создать интерфейс для новой Web-службы. Ответ на этот вопрос зависит от того, что именно вы собираетесь публиковать. Если вы проводите первые эксперименты с SOAP, без раздумий отвечайте "да", тем более, что далее Borland предложит вам полезный пример — наше старое знакомое "эхо". Если вы доросли до реального приложения, к этому моменты вы уже будете знать, что и как вам надо опубликовать. О вариантах публикации позже, а пока продолжим создание нашей первой Web-службы.

После выбора типа приложения среда разработки запрашивает программиста о необходимости создания интерфейса для модуля SOAP. При утвердительном ответе появляется диалог создания новой Web-службы, который выглядит, как показано на рис. 40.1. Имя службы — идентификатор, под которым служба

будет известна клиентам. Логично, чтобы создаваемые модули Delphi имели тот же идентификатор. Если точнее, то модулей будет создано два, в данном случае `SampleServiceInft.pas` и `SampleServiceImpl.pas`.

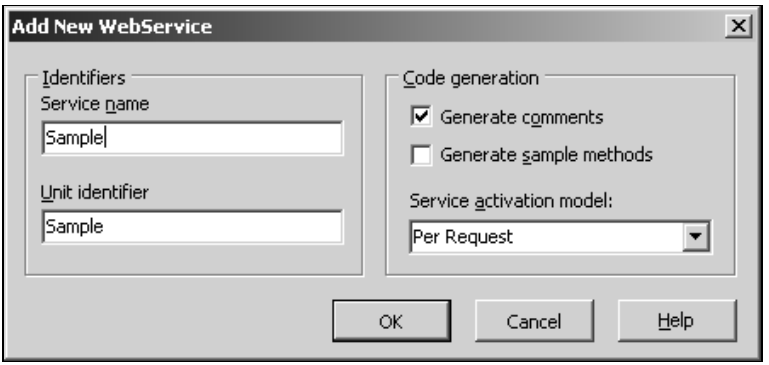


Рис. 40.1. Диалоговое окно параметров интерфейса Web-службы

Почему и зачем? Суффиксы `Intf` (Interface) и `Impl` (Implementation) приводят нас к ответу. Интерфейсная часть службы понадобится как серверу, так и клиенту (если, разумеется, этот клиент написан на Delphi, в ином случае достаточно WSDL-файла). Именно поэтому интерфейс в примере отделен от кода объекта, реализующего (implementing) этот интерфейс.

Примечание

Разумеется, и для клиента, написанного на Delphi, достаточно WSDL-файла — иначе как бы такие клиенты соединялись со службами других поставщиков? Но мастер импорта WSDL создаст для вас почти точно такой же файл интерфейса, какой уже имеется на сервере (в данном примере — аналог `SampleServiceInft.pas`). Так зачем же два раза делать одну и ту же работу?

Флажок **Generate sample methods** включит в сгенерированный интерфейс примера четыре метода:

```
  ISampleService = interface(IInvokable)
  ['{A5F9E9F3-4781-4465-900D-EC2BE6E17219}']
    { Methods of Invokable interface must not use the default }
    { calling convention; stdcall is recommended }
    function echoEnum(const Value: TEnumTest): TEnumTest; stdcall;
    function echoDoubleArray(const Value: TDoubleArray): TDoubleArray;
stdcall;
```

```
    function echoMyEmployee(const Value: TMyEmployee): TMyEmployee;
stdcall;
    function echoDouble(const Value: Double): Double; stdcall;
  end;
```

Префикс `echo` напоминает название соответствующего коммуникационного протокола, и это действительно то же самое — все четыре метода отправляют назад в качестве результата полученное входное значение. Почему методов четыре, а не один? Потому что четыре различных типа параметров иллюстрируют особенности обмена разными — сложными — типами данных. Опыт обмена такими данными пригодится нам в дальнейшем.

После добавления интерфейса перед нами — созданный Web-модуль с тремя расположенными на нем компонентами (рис. 40.2).

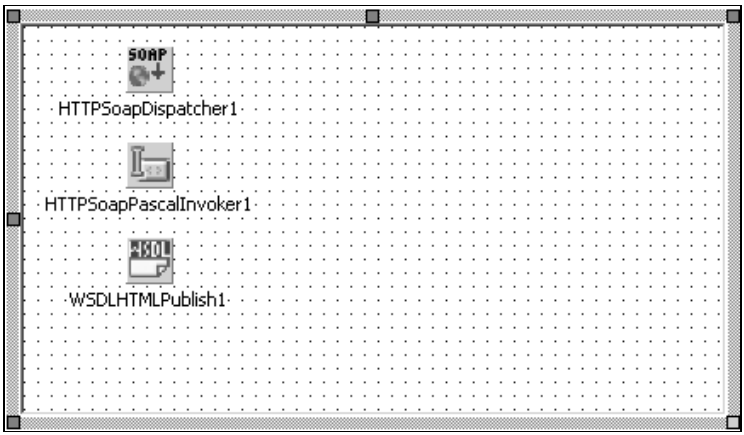


Рис. 40.2. Web-модуль, создаваемый мастером SOAP Server Application

Сервер в его простейшем виде "готов к употреблению". Скомпилируем нашу DLL (в случае ISAPI) и поместим ее в один из каталогов Web-сервера. Надо только помнить, что у него должны быть установлены разрешения на запуск сценариев и исполняемых файлов. Если вы размещаете приложение в каталоге scripts, который автоматически создается IIS и предназначен как раз для этих целей, то проблем нет. Если же вы решили отвести под файлы службы отдельный каталог, не забудьте настроить права доступа.

Важное примечание

В последнее время появилось много вирусов, использующих уязвимые места в IIS. Проникая на сервер, они запускают на нем свой код, заражая компьютер и распространяясь по сети. И деструктивное действие, и скорость их распространения могут быть чрезвычайно велики. Так, только во время написания

этой главы наш отладочный сервер был атакован три раза. Поэтому если ваш сервер соединен с Internet, и о безопасности не позаботился ваш системный администратор (или у вас его нет), обязательно посетите сайт Microsoft, посвященный проблемам безопасности, HotFix & Security Bulletin Service, по адресу **http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/current.asp**.

В частности, критически важно установить свежий пакет обновления (hot fix). Без этого работу начинать опасно.

Назначение и настройки компонентов серверной части

Перед тем как переходить к рассмотрению более сложных вопросов, нужно разобраться, какую же роль играют увиденные нами компоненты (см. рис. 40.2).

Компонент `TWebModule`, как известно, является централизованным диспетчером и обработчиком запросов HTTP со стороны клиента. Действия, на которые он реагирует, сосредоточены обычно в одноименном свойстве `Actions`. Но есть и исключения, точнее, дополнения к этому правилу — так называемые компоненты с возможностью автодиспетчеризации. Проще всего рассматривать их как "внешние" по отношению к диспетчеру действия. У них есть те же свойства `MethodType` и `PathInfo`, которые позволяют диспетчеру перенаправить поступивший запрос.

Для того чтобы зарегистрировать компоненты автодиспетчеризации и связанные с ними действия, достаточно просто сделать их дочерними компонентами некоторого диспетчера. Самый простой способ для этого — поместить их на компонент `TWebModule`.

В данном случае модуль передает действия для диспетчеризации двум компонентам: `THTTPSoapDispatcher` и `TWSDLHTMLPublish`.

В предыдущей главе мы рассматривали два действия, которые характерны для служб, написанных в Delphi. Первое — `/soap` — давало доступ непосредственно к интерфейсам службы. Второе — `/wsdl` — позволяло получить файл ее описания. Очевидно, таково и назначение двух упомянутых компонентов.

Общим для них (и для всех компонентов с автодиспетчеризацией) является свойство `WebDispatch`, задающее правила диспетчеризации. Его тип в свою очередь является классом с такими основными свойствами:

```
property PathInfo: string;
property MethodType: TMethodType;
property Enabled: Boolean;
```

Практической пользы из этих свойств извлечь можно немного. Значение `MethodType` предопределено стандартом SOAP: для запроса WSDL-информации это метод `HTTP GET`, для обращения к портам — `POST` или `M-POST`. Соответственно, вы увидите в этом свойстве значения `mtAny` для `TWSDLHTMLPublish` и `mtPost` — для `THTTPSoapDispatcher`.

Свойство `PathInfo` пригодится вам, если вы по каким-то причинам решите переименовать действие. Скажем, если вы захотите переименовать часть URL, отвечающую за путь к службам, с `/soap` на `/shampoo`, вы можете отредактировать значение `PathInfo`.

Компонент *TWSDLHTMLPublish*

Длинное название становится простым, если разбить его на составные части: WSDL+HTML+Publish. Совершенно верно, он публикует описания службы в удобочитаемом формате HTML-файла. Самодокументируемость интерфейсов стала "правилом хорошего тона" для разработчиков SOAP-приложений. Если вы по каким-то причинам не хотите публиковать документацию вашей службы, то можете спокойно убрать экземпляр `TWSDLHTMLPublish` из Web-модуля — на работу остальных составных частей это ровным счетом никак не повлияет. При этом надо полагать, что WSDL-файл службы вы сгенерируете и доставите клиенту иным известным вам способом.

Функциональность этого компонента достаточно проста — он публикует *все* содержимое Invokation registry, для этого достаточно одного факта присутствия `TWSDLHTMLPublish`. Поэтому беспокоиться о настройке компонента на новые и добавленные интерфейсы не стоит — если они корректно зарегистрированы, то автоматически появятся в публикуемой документации.

Еще раз напомним, что действие с адресом `/wsdl` (в нашем случае **http://localhost/scripts/simpleecho.dll/wsdl**) выводит страницу описания интерфейсов (рис. 40.3).

> **Примечание**
>
> Код Web-модуля, генерируемого мастером, переадресует на `TWSDLHTMLPublish` еще и действие по умолчанию (осуществляемое при отсутствии дополнительных символов после имени модуля ISAPI или CGI). То есть в нашем случае попасть на страницу описания службы можно и набрав просто **http://localhost/scripts/simpleecho.dll**.

Из каких колонок состоит таблица на рис. 40.3? Вы можете найти сходство между ними и параметрами метода `RegisterInterface` глобального объекта `InvokeRegistry`. В частности, в параметрах этого метода можно указать краткое описание интерфейса (параметр `Doc`).

Исправив стандартный код регистрации (сгенерированный мастером) на

```
InvRegistry.RegisterInterface(TypeInfo(ISimpleEcho),
'urn:SimpleEchoIntf-ISimpleEcho', '',
'Test interface: just returns the input parameter');
```

мы сможем увидеть это описание в таблице (см. рис. 40.3).

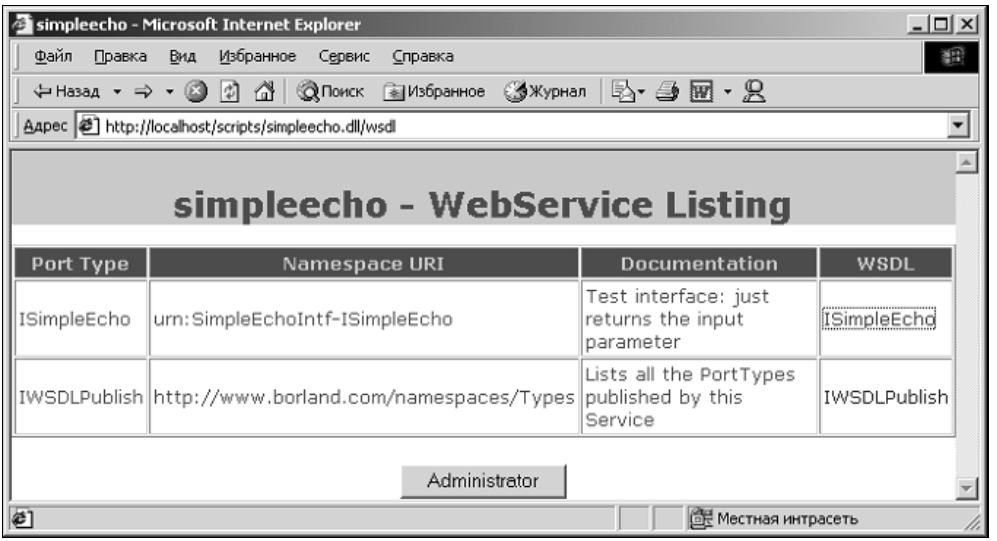


Рис. 40.3. HTML-документ с описанием портов службы, генерируемый компонентом `TWSDLHTMLPublish`

Рассматриваемый компонент позволяет решить еще одну важную задачу — выполнить динамическую настройку WSDL-файла. Предположим, вы создавали и редактировали службу в условиях отладки, а теперь переносите ее с отладочного на рабочий сервер. Самое время поменять адреса портов в WSDL, но как это сделать? На помощь приходит свойство

```
Property AdminEnabled: Boolean;
```

которое разрешает удаленное редактирование WSDL при помощи браузера. Если оно содержит значение `True`, под списком интерфейсов появляется кнопка **Administrator**, которая видна на рис. 40.3.

Нажав кнопку, вы попадаете в простой редактор портов (рис. 40.4). Собственно, больше ничего на этом этапе поменять нельзя, т. к. типы портов и форматы сообщений должны были быть стандартизованы заранее. Переменной частью в данном случае является только имя и адрес порта, который вы и можете добавить, удалить или изменить.

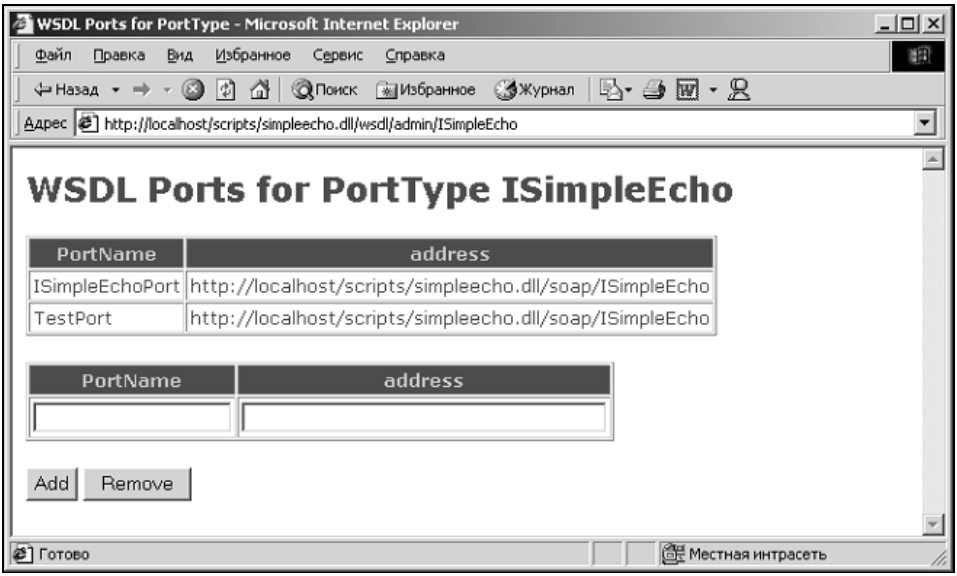


Рис. 40.4. Администрирование списка портов в браузере

В примере на рис. 40.1 мы добавили еще один порт с именем `TestPort`. Возвратившись назад и снова просмотрев содержимое WSDL-документа, мы увидим в нем этот порт.

Как осуществляется запоминание введенных портов, ведь, как мы помним, WSDL-документ формируется динамически? Запись параметров администрирования осуществляется в файл с именем <имя службы>_wsdladmin.ini, который располагается в той же папке, где и сама служба. При каждом обращении к описанию компонент `TWSDLHTMLPublish` просматривает этот файл и добавляет порты к документу. Настроив их, вы можете перекомпилировать приложение и отключить режим администрирования, присвоив свойству `AdminEnabled` значение `False`. При этом INI-файл с отредактированным списком портов не исчезнет и по-прежнему будет отображаться в соответствии с настройками.

Примечание

Запись параметров происходит в INI-файл из-за ограничений на использование реестра, характерных для Web-приложений. Это следует взять на заметку и вам. Если ваша служба имеет любые настраиваемые параметры, то вам придется запоминать их не в реестре, а в соответствующем INI-файле.

Параметр с длинным названием `poPublishLocationAsSecure` (свойство `PublishOptions`) позволяет вставить одну лишь, но очень важную, букву в короткое название протокола. Поскольку вручную изменить файл описания

нельзя, разработчику нужен способ указания пользователю службы на то, что связь будет осуществляться в защищенном режиме посредством SSL. Присвоив этому параметру `True`, вы добиваетесь того, что адреса всех портов, указывающих на данную службу, будут содержать в адресе https:// вместо http://.

Значение еще одного свойства

```
property TargetNamespace: WideString;
```

включается в корневой элемент `<definition>` WSDL-файла службы и может служить частью ее описания.

Компоненты *THTTPSoapDispatcher* и *THTTPSoapPascalInvoker*

Эта пара отвечает за трансляцию посылок SOAP в вызовы методов invokable interfaces (вызываемых интерфейсов).

Первый компонент играет роль *согласователя*. Принимая на вход запросы HTTP, он осуществляет их буферизацию, перекодировку и извлечение конверта SOAP, который он отправляет связанному с ним компоненту типа `THTTPSoapPascalInvoker`. После получения ответа происходит обратное преобразование.

Ни сколько-нибудь важных свойств, ни новых опубликованных событий у него нет; важен сам факт присутствия `THTTPSoapDispatcher` в Web-модуле.

С точки зрения разработчика, намного важнее `THTTPSoapPascalInvoker`, отвечающий за вызов методов.

Во-первых, вы можете проконтролировать процесс вызова методов интерфейса, описав обработчики трех событий:

```
property BeforeDispatchEvent: TBeforeDispatchEvent;
TBeforeDispatchEvent = procedure(const MethodName: string; const Request:
TStream) of object;
property AfterDispatchEvent: TAfterDispatchEvent;
TAfterDispatchEvent = procedure(const MethodName: string; SOAPResponse:
TStream) of object;
property OnExceptionEvent: TOnExceptionEvent;
TOnExceptionEvent = procedure(const MethodName: string; const Request:
TStream; const Response: TStream) of object;
```

Во-вторых, этот компонент имеет набор параметров (свойство `Options`), определяющий совместимость.

Клиент службы *SimpleEcho*

Для тестирования нашей службы нужна теперь клиентская часть. Возьмем для этого с минимальными переделками клиента из рассмотренной в предыдущей главе службы BASE, тем более что по функциональности это просто "братья-близнецы" (рис. 40.5, код приводится на компакт-диске, прилагаемом к книге).

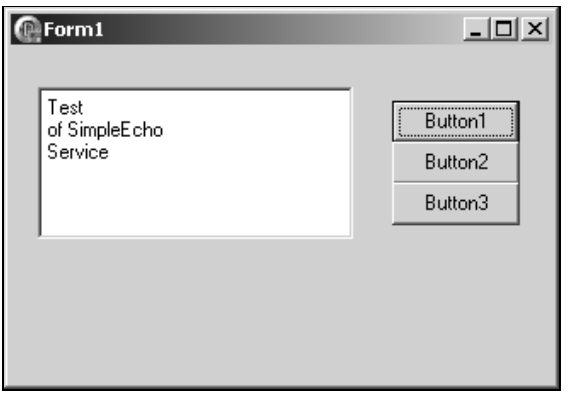


Рис. 40.5. Главное окно клиента службы `SimpleEcho`

Проделаем только одну неочевидную вещь. Для описания службы на клиентской стороне нужен файл описания интерфейса службы. Можно взять уже готовый файл SimpleEchoIntf.pas, сгенерированный SOAP Server Application Wizard, и включить его в клиентский проект.

Но можно пойти и другим путем — снова вызвать мастер импорта WSDL-файлов и создать описание интерфейса службы с его помощью. Напомним, что различие будет минимальным: второй вариант дополнительно предоставляет нам функцию `GetISimpleEcho`. Поскольку она нам понадобится, исключим SimpleEchoIntf.pas из проекта сервера и включим вместо него клиентский файл. Назовем его для разнообразия IntfSimpleEcho.pas.

Обработка исключительных ситуаций

Если вы как следует поняли логику вызываемых (Invokable) объектов и интерфейсов, то сейчас легко оцените принципы возбуждения и обработки исключительных ситуаций (exceptions) в Web-службах в Delphi. Для этого достаточно только вспомнить, что исключительные ситуации (ИС) являются такими же полноправными классами, как и все прочие.

Класс, подлежащий маршаллингу посредством протокола SOAP, должен быть скомпилирован с RTTI. Мы уже сказали о том, что интерфейсы долж-

ны быть порождены от `IInvokable`, а классы Delphi — от `TRemotable`. Для исключений родительский класс называется `ERemotableException`. Все порожденные от него классы исключительных ситуаций могут передаваться клиенту от сервера, если, разумеется, они правильно зарегистрированы.

Давайте создадим наш класс исключительной ситуации. Алгоритм действий таков.

1. Описываем собственную ИС (для нашего примера мы создадим `EEchoException`). Лучшее место для ее описания и регистрации — модуль описания интерфейса IntfSimpleEcho.pas; он, как вы помните, совместно используется в проектах и сервера, и клиента.

   ```
   EEchoException = class(ERemotableException);
   ```

2. Регистрируем ИС при помощи вызова метода `RegisterXSClass` глобального объекта `RemTypeRegistry`:

   ```
   RemTypeRegistry.RegisterXSClass(EEchoException);
   ```

   Теперь оба проекта будут "признавать" этот класс.

3. В коде сервера в нужный момент возбуждаем ИС. Исправим код метода `echoString` (модуль `SimpleEchoImpl.pas`) следующим образом:

   ```
   function TSimpleEcho.echoString(const Value: String): String;
   begin
     if AnsiCompareStr(Value, 'Error') = 0 then
      raise EEchoException.Create(Value)
     else
      Result := Value;
   end;
   ```

4. Возбудив ИС, мы "отдали пас" библиотеке времени исполнения Delphi. Обнаружив `EEchoException`, она делает следующее:

   - проверяет, зарегистрирована ли данная ИС в реестре Remotable классов (если нет, будет возбуждаться `ERemotableException`);
   - упаковывает свойства класса ИС в так называемый fault-пакет (присылается службой в ответ на запрос клиента, если на серверной стороне произошла та или иная ошибка);
   - прервав действие текущего метода интерфейса службы, отправляет пакет клиенту.

5. На клиентской стороне библиотека времени исполнения проверяет ответ службы. В случае получения fault-пакета генерируется ИС, для зарегистрированных ИС возбуждается "правильный" (присланный) класс ИС, для неизвестных — возбуждается `ERemotableException`. Свойства этой ИС заполняются содержимым fault-пакета.

6. Итак, мы получили обратный "пас" в виде возбужденной исключительной ситуации. Дело разработчика клиентской части — правильно обработать ее. Модифицированный код обработчика нажатия кнопки `Button2` в клиентском приложении выглядит так:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 try
  ShowMessage(
   (HTTPRIO1 as ISimpleEcho).echoString(meMessage.Lines[0]));
 except
  on E:EEchoException do
   begin
    ShowMessage('Must not contain string: ' + E.Message + #13#10
    + E.FaultActor + #13#10 + E.FaultCode + #13#10+ E.FaultDetail);
   end;
 end;
end;
```

Вы заметили у обнаруженной ИС три свойства: `FaultActor`, `FaultCode` и `FaultDetail`. Они соответствуют трем элементам fault-пакета, стандартизованных в SOAP. Даже если клиент вашей службы написан не в Delphi, он будет в состоянии прочитать информацию о происшедшем. Вот пример пакета, посылаемого в описанном выше случае:

```
<?xml version="1.0" ?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <SOAP-ENV:Fault>
   <SOAP-ENV:faultcode>SOAP-ENV:Server</SOAP-ENV:faultcode>
   <SOAP-ENV:faultstring>Error</SOAP-ENV:faultstring>
   <NS1:detail xmlns:NS1="urn:intfsimpleecho"
xsi:type="NS1:EEchoException" />
  </SOAP-ENV:Fault>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Вы можете переопределять значения свойств при возбуждении исключительной ситуации; более того, вы можете не ограничиваться тремя стандартными свойствами. Добавленные вами свойства (к классу ИС, порожденному

от `ERemotableException`), будут также упакованы и переданы на клиентскую сторону.

Кроме ваших собственных ИС, позаботьтесь об обработке `ESOAPDomConvertError`. Эта ошибка возбуждается внутри компонентов `TOPToSoapDomConvert` (напомним, они есть в составе других компонентов и на серверной, и на клиентской стороне). Причина ее возникновения — любая проблема *сериализации (serialization)*, т. е. отображения пакетов SOAP на вызовы функций и обратно. Этой ИС можно ждать при любых отклонениях от стандарта SOAP, что, увы, не редкость.

Средства разработки для SOAP: подход Microsoft

Было бы странно писать книгу о новейших решениях для обеспечения кроссплатформенной совместимости между приложениями и рассмотреть только конкретное инструментальное средство одной фирмы для единственной платформы. С другой стороны, полный анализ всех предлагаемых программных решений явно выходит за рамки этой книги. Поэтому мы ограничимся еще одним инструментом для работы со службами и на примере их взаимодействия оценим круг возможных задач и проблем совместимости. Это средство — Microsoft SOAP Toolkit.

Фирма Microsoft предложила средства разработки для SOAP одной из первых. Вначале появился SOAP Toolkit версии 1.0, использовавший устаревший ныне язык описания запросов SDL. Он изначально предназначался для тестирования новой технологии, подготовки к переходу на архитектуру .NET и полноценным продуктом так и не стал. Новая версия (2.0) применяла уже стандартный язык WSDL и существенно отличалась улучшенной производительностью и удобством использования.

Итак, почему выбор пал на MS SOAP Toolkit в качестве примера другого инструментального средства для работы с SOAP и Web-службами? Во-первых, SOAP Toolkit является свободно распространяемым продуктом, но, тем не менее, полноценно поддерживается Microsoft. Во-вторых, он может быть реализован почти на всех современных платформах Windows с незначительными ограничениями и не предъявляет особых требований при установке. В-третьих, он тесно увязан с моделью COM, что для наших читателей является только преимуществом. И наконец, его архитектура весьма напоминает подход, примененный в Delphi.

Для работы с клиентской частью toolkit нам потребуется рассмотреть короткие сценарии, написанные на VBScript. Мы уверены, что они не покажутся сложными даже разработчикам, впервые сталкивающимся с этим языком.

Средства для интерпретации сценариев — утилиты `cscript` или `wscript` — вы найдете практически на любом компьютере с Windows.

Краткий обзор архитектуры Web-служб в SOAP Toolkit

Прежде чем приступать к разработке средствами SOAP Toolkit, давайте бросим взгляд на общий алгоритм его работы и пути потоков данных (dataflow) на серверной и клиентской стороне.

1. Собственно службами, предоставляющими свои интерфейсы, являются COM-объекты. В Delphi объект, применяющийся в качестве службы, должен поддерживать интерфейсы — потомки `IInvokable`. В SOAP Toolkit требование таково: объект должен поддерживать `IDispatch`, т. е. быть объектом Автоматизации. Как правило, это внутренний (in-process) сервер — динамическая библиотека (Dynamic Link Library, DLL); допускается применение и EXE-серверов.

   В состав toolkit входит утилита WSDL Generator, которая строит файл описания службы для объекта Автоматизации. Вы выбираете любой объект из числа зарегистрированных на данном компьютере и получаете требуемое описание. Помехой здесь могут стать нестандартные типы данных; для интерфейсов, методы которых обмениваются объектами или структурами, понадобятся дополнительные действия.

   Особенностью инструмента MS SOAP Toolkit является генерация не одного (WSDL), а сразу двух файлов — WSDL и WSML. Первый является обычным стандартным файлом описания, соответствующим спецификации WSDL 1.1, а вот второй (название произошло от сокращения Web Services Meta Language) специфичен. Он играет ту же роль, что и Invokation registry в Web-службах, написанных в Delphi. В таком файле производится сопоставление порта службы с конкретным COM-объектом (задаваемым своим `ProgID`), а операции порта отображаются на вызовы методов объекта:

```
<servicemapping name='SimpleEcho'>
 <service name='SimpleEcho'>
  <using PROGID='SimpleEchoCOM.SimpleEcho' cachable='0'
  ID='SimpleEchoObject' />
  <port name='SimpleEchoSoapPort'>
    <operation name='echoStringArray'>
      <execute uses='SimpleEchoObject' method='echoStringArray'
      dispID='2'>
        <parameter callIndex='1' name='Value' elementName='Value' />
        <parameter callIndex='-1' name='retval' elementName='Result'
/>
```

```
      </execute>
    </operation>

  </port>
 </service>
</servicemapping>
```

2. За вызовы COM-объектов отвечают специальные модули расширения для Web-сервера, которые называются *слушателями (listener)*. В состав toolkit входят два слушателя: ISAPI-расширение и файл ASP. Получив запрос SOAP от Web-сервера, они интерпретируют WSDL- и WSML-файлы и создают специальный объект `SOAPServer` (или `SOAPServer30` в версии 3.0). Он выполняет те же функции, что и объект класса `TSoapPascalInvoker`, — вызывает собственно методы запрашиваемого COM-объекта.

   Достоинства первого слушателя являются недостатками второго и наоборот. ISAPI-слушатель намного быстрее, он кэширует создаваемые объекты. Но при этом он не позволяет делать ни малейшей настройки на серверной стороне.

   ASP-слушатель создается в виде исходного файла на языке сценариев. Разумеется, интерпретируемый язык, несмотря на оптимизацию, не может составить конкуренцию по скорости коду динамической библиотеки ISAPI. Зато весь исходный текст перед вами, и вы можете вставить туда дополнительную проверку и преобразование параметров, инициализацию переменных и объектов, механизм управления сессиями и т. п.

   Более того, ASP-слушатель может вообще не вызывать никаких внешних объектов. Пользователям SOAP Toolkit предоставляются два API: высокоуровневый и низкоуровневый. Второй позволяет разработчику анализировать SOAP-запросы и форматировать ответы; таким образом, весь необходимый код службы может быть написан на языке сценариев и находиться прямо в ASP-файле.

   Адрес порта, характерный для MS SOAP Toolkit, может показаться читателю несколько необычным:

```
<definitions>
<service name='SimpleEcho' >
 <port name='SimpleEchoSoapPort' bind-
ing='wsdlns:SimpleEchoSoapBinding' >
    <soap:address location='http://localhost/scripts/SimpleEcho.wsdl'
/>
 </port>
</service>
</definitions>
```

Как видим, адрес порта в WSDL указывает непосредственно на WSDL, который, как вы понимаете, не содержит и не может содержать никакого исполняемого кода. Дело в том, что файлы WSDL располагаются в специально созданных Web-папках IIS. Для этих папок слушатель регистрируется как средство, которое будет вызываться при обращении к документам WSDL в этой папке. При этом ISAPI-слушатель может находиться вне папки и вообще за пределами дерева публикации IIS (обычно C:\InetPub). Это помогает решать проблемы с настройками и безопасностью.

Итак, снова подчеркнем разницу между двумя подходами. В Delphi каждая служба представляет собой модуль расширения для Web-сервера (ISAPI или CGI); вся функциональная начинка может находиться в этом же модуле. SOAP Toolkit использует один общий модуль расширения — слушатель. Слушатель вызывает методы разнообразных COM-объектов, которые обеспечивают функционирование службы.

3. Клиентская сторона очень похожа на то, что мы уже рассматривали в Delphi. Роль `THTTPRIO` здесь исполняет объект под названием `SOAPClient` (`SOAPClient30`).

   Первым делом после создания нужно инициализировать его, заставляя "притвориться" портом службы:

```
Call soapclient.mssoapinit(_
    "http://localhost/scripts/SimpleEcho.dll/wsdl/ISimpleEcho", _
    "ISimpleEchoservice",_
    "ISimpleEchoPort")
```

   После вызова и успешного завершения метода `mssoapinit` объект `SOAPClient` готов к работе: вы можете вызывать операции порта (они же — методы исходного COM-объекта на серверной стороне), как собственные методы `SOAPClient`.

Сценарий клиентской части

Для начала создадим простого клиента, который бы соединялся с нашей службой SimpleEcho. Для этого мы будем использовать Microsoft SOAP Toolkit версии 2. К моменту написания книги версия 3 находилась на стадии первого бета-релиза, в то время как к версии 2 вышел уже второй пакет обновления. Кроме того, версия 3 содержит XML-анализатор Microsoft версии 4, работа с которым требует переделки некоторых модулей библиотеки Delphi *(см. об этом в главе 39)*. Скачать Microsoft SOAP Toolkit 2.0 SP2 вы можете по адресу **http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/001/580/msdncompositedoc.xml**.

Системные требования при этом довольно просты. Для серверных объектов это Windows NT 4.0 Service Pack 6, Windows 2000 или XP; соответственно для этих ОС, IIS 4, 5.0 и 5.1. Для клиента достаточно любой версии Windows старше, чем Windows 95. И там, и там должен быть установлен Internet Explorer 5 или выше.

Единственным неприятным ограничением является то, что для работы утилиты WSDL Generator (wsdlgen.exe) необходимо наличие на компьютере среды разработки Microsoft Visual Studio 6. Впрочем, это требование можно обойти, использовав взамен аналогичную утилиту, работающую в режиме командной строки — wsdlstb.exe.

Код клиента содержит чуть больше двух десятков строк. Создайте файл SimpleEchoCli.VBS и поместите в него текст из листинга 40.1 (впрочем, этот файл есть среди примеров на диске, поставляемом с книгой).

Листинг 40.1 Код клиента службы SimpleEcho с использованием SOAP Toolkit (на языке VBScript)

```
Sub CallEcho(ByVal Text)
Dim soapclient
set soapclient = CreateObject("MSSOAP.SoapClient")
On Error Resume Next
Call soapclient.mssoapinit(_
  "http://localhost/scripts/SimpleEcho.dll/wsdl/ISimpleEcho", _
  "ISimpleEchoservice",_
  "ISimpleEchoPort")
if err <> 0 then
  wscript.echo "Initialization failed " + err.description
  Exit Sub
end if
wscript.echo  soapclient.echoString(Text)
if err <> 0 then
  WScript.echo   err.description
  WScript.echo   "faultcode=" + soapclient.faultcode
  WScript.echo   "faultstring=" + soapclient.faultstring
  WScript.echo   "faultactor=" + soapclient.faultactor
  WScript.echo   "detail=" + soapclient.detail
end if
End Sub
rem ----------------------------------------------------
Dim Text
```

```
If WScript.Arguments.Count<1 Then
 Text = "Hello, World!"
Else
 Text = WScript.Arguments(0)
End If
Call CallEcho(Text)
```

Внутри процедуры `CallEcho` происходит инициализация объекта `SOAPClient` — он настраивается на WSDL, генерируемый нашим Delphi-сервером службы SimpleEcho (**http://localhost/scripts/SimpleEcho.dll/wsdl/ISimpleEcho**). При наборе следующих значений параметров — имени службы и порта — обращайте внимание на регистр букв; в отличие от Delphi здесь он играет роль.

Если инициализация завершилась нормально, то сценарий вызовет метод `echoString` с переданным в процедуру текстом. Текст представляет собой параметр командной строки сценария (если он был задан) или традиционное "Hello, World!"

В случае получения результата `fault` (а такое, напомним, произойдет, если входная строка содержала текст 'Error') подробная информация о происшедшем — параметры `faultcode`, `faultstring`, `faultactor` и `detail` — будет показана пользователю.

Примечание

Вызов метода `echoStringArray` требует ручной корректировки WSDL, т. к. язык сценариев VBScript обладает весьма специфической обработкой массивов. Отметьте для себя эту проблему несовместимости; если вам нужна более подробная информация о способах решения этой проблемы, обратитесь к документации MS SOAP Toolkit.

Вы можете запустить сценарий с помощью утилит `wscript` (выводит сообщения в виде диалоговых окон Windows) или `cscript` (выводит их в стандартный файл вывода, как правило, окно текущей сессии командного интерпретатора).

Создание COM-объекта *SimpleEchoCOM* и публикация Web-службы на его основе

Теперь возьмемся за серверную часть службы. Первым делом нам понадобится объект Автоматизации, интерфейсы которого будут вызываться слушателем SOAP Toolkit.

Как вы могли прочитать в предыдущей главе, за интерфейсами Web-служб Delphi, порожденными от `IInvokable`, не стоят реальные COM-объекты.

Поэтому объект Автоматизации нам придется создавать "с нуля". Алгоритм действий таков.

1. В Репозитории объектов Delphi (меню **File | New | Others**) выберите новый проект типа **ActiveX Library** и сохраните его с именем SimpleEchoCOM в папке под названием COMServer. Этот внутренний сервер будет носителем нужного объекта. (Если считаете повторение этого и нижеследующих шагов излишним, готовый сервер вы найдете в сборнике примеров на прилагаемой дискете.)
2. Вернувшись на ту же страницу Репозитория, выберите пункт **New Automation Object**. Назовите новый объект `SimpleEcho`, тогда к библиотеке добавится интерфейс `ISimpleEcho`, порожденный от `IDispatch`. Сохраните файл реализации объекта как `SimpleEchoCOMImpl.pas` (чтобы не путать с уже существующим `SimpleEchoImpl.pas`).
3. Добавьте в библиотеке типов два метода к интерфейсу `ISimpleEcho`, как показано на рис. 40.6.

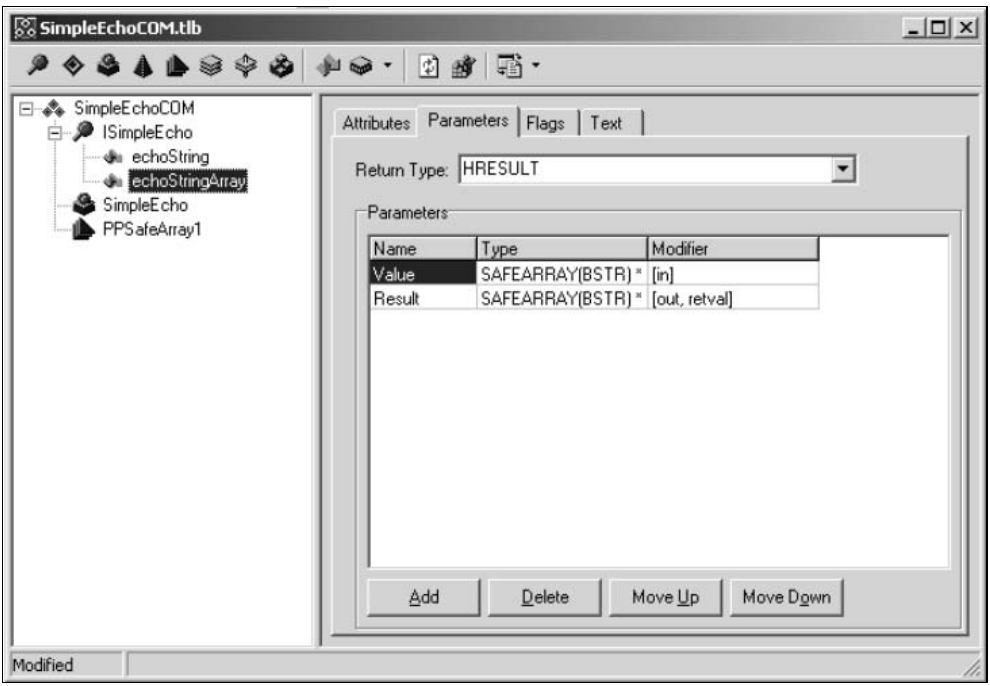


Рис. 40.6. Внешний вид библиотеки типов проекта SimpleEchoCOM

Массивы строк, передаваемые в метод `echoStringArray`, должны иметь тип `SAFEARRAY(BSTR)*` (мы не можем применить класс `TStringDynArray`,

т. к. динамические массивы Delphi не поддерживаются в Автоматизации; ближайший аналог — массивы типа `SAFEARRAY`).

4. Добавьте код методов `echoString` и `echoStringArray`, он чрезвычайно прост:

```
function TSimpleEcho.echoString(const Value: WideString): WideString;
begin
  Result := Value;
end;
function TSimpleEcho.echoStringArray(var Value: PSafeArray):
PSafeArray;
begin
  SafeArrayCopy(Value, Result);
end;
```

5. Скомпилируйте, сохраните и зарегистрируйте (меню **Run | Register ActiveX Server**) созданный объект.

Теперь можно переходить к созданию и публикации WSDL-файла для новой службы, точнее, старой службы в новом обличье.

Если у вас не установлен Microsoft Visual Studio, запустите утилиту wsdlstb.exe, указав ей в командной строке путь к COM-объекту, местоположение ASP-слушателя и папку, куда вы хотите поместить создаваемые WSDL- и WSML-файлы. Сохраним их в папке scripts, где уже находится файл simpleecho.dll. Если вы хотите использовать ISAPI-слушатель (а это в данном случае разумнее), в сгенерированном файле надо будет заменить значение атрибута `location` порта службы на **http://localhost/scripts/SimpleEcho.wsdl**.

В процессе выполнения утилита попросит вас ввести имя интерфейса (у нас он один — `ISimpleEcho`) и имена методов, которые должны попасть в описание (включите оба — `echoString` и `echoStringArray`).

Сгенерированный файл весьма похож на тот, который выдается сервером, написанным в Delphi. Отличие есть в названии службы и порта (`SimpleEcho` вместо `ISimpleEchoservice` и `SimpleEchoSoapPort` вместо `isimpleechoPort`). Соответственно, придется внести небольшую правку в WSDL и присвоить названиям старые значения. Можно пойти другим путем и изменить параметры соединения как Delphi-, так и VBScript-клиента.

Теперь в нашем распоряжении два различных сервера и два клиента. Этого вполне достаточно, чтобы провести хотя бы минимальные эксперименты по обеспечению совместимости (interoperability). Рассмотренный нами пример, исходный код которого вы найдете на прилагаемом к книге носителе, обес-

печивает полную совместимость с учетом единственного исключения: VBScript-клиент не содержит вызова метода `echoStringArray`.

Основные проблемы совместимости, как правило, порождаются несоответствием типов данных. В связи с этим логично вернуться к службе BASE *(см. главу 39)*, которая, как мы помним, является эхом не для двух, а для двух десятков типов данных. Вопросы совместимости здесь наверняка возникнут. И тут нам на помощь приходит другая утилита из состава MS SOAP Toolkit, предназначенная для отладки служб.

Использование утилиты SOAP Trace

Еще одна причина для того, чтобы рассмотреть в этой книге MS SOAP Toolkit — наличие в нем чрезвычайно полезной утилиты SOAP Trace (MSSOAPT.EXE). Она перехватывает трафик HTTP и записывает его в журнал. Но в отличие от ряда аналогичных утилит, она умеет не просто перехватывать запросы (request) и ответы (response) HTTP, но еще и декодирует пакеты SOAP. Таким образом, вы можете просмотреть, что именно вы отослали серверу, и как выглядит на самом деле его ответ.

Переоценить полезность SOAP Trace трудно. Пока технологии и стандарты Web-служб активно развиваются, шероховатости неизбежны. Просмотр сообщений SOAP поможет локализовать проблему.

Технически перехват трафика осуществляется очень просто. Утилита выступает в роли *"подставного" (fake)* сервера. WSDL-файлы переделываются таким образом, чтобы указывать на адрес хоста и порт, на котором утилита "слушает" входящий трафик. Получив и запротоколировав посылку, SOAP Trace отправляет ее "настоящему" серверу.

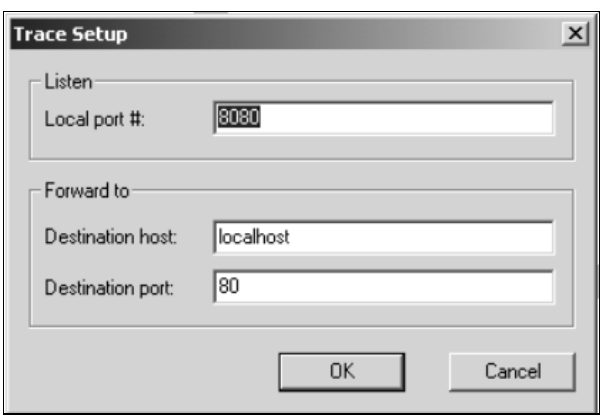


Рис. 40.7. Настройка параметров утилиты SOAP Trace

Диалоговое окно настройки параметров перехвата выглядит как на рис. 40.7.

Пользоваться SOAP Trace можно как на серверной, так и на клиентской стороне.

Для использования ее на стороне сервера в WSDL-файле достаточно просто заменить номер порта 80 (это, напомним, порт HTTP по умолчанию) на 8080, например: **http://soap.myservices.com/scripts/** на **http://soap.myservices.com:8080/scripts/**.

Примечание

Разумеется, следует поинтересоваться, открыт ли порт 8080, и в противном случае выбрать один из доступных неиспользуемых портов.

В этом случае при запуске утилиты настроек, показанных на рис. 40.7, вообще менять не надо: пакеты будут переадресовываться с адреса localhost:8080 на localhost:80, а это именно то, что требуется.

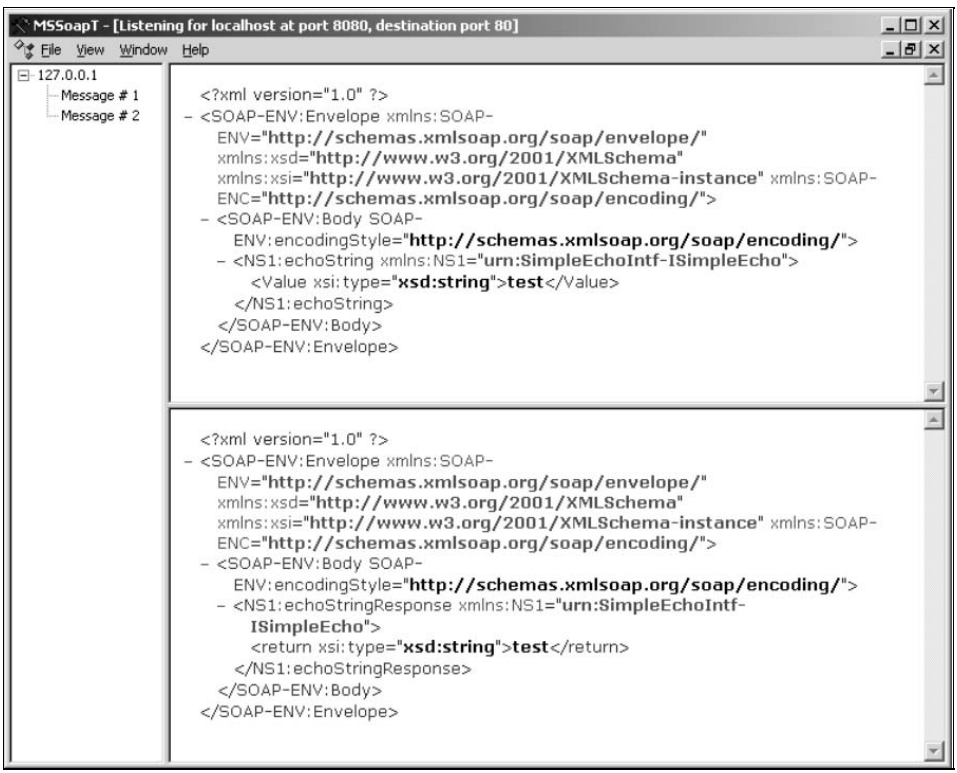


Рис. 40.8. Утилита SOAP Trace — просмотр в режиме форматированного вывода

При запуске SOAP Trace на клиентской стороне необходимо чуть больше шагов. Во-первых, WSDL-файл следует переделать таким образом, чтобы он указывал на localhost:8080. В приведенном примере **http://soap.myservices.com/scripts/** следует поменять на **http://localhost:8080/scripts/**.

Во-вторых, при вводе параметров утилиты в поле **Destination host** нужно занести истинный адрес сервера — **soap.myservices.com**. Обращаем ваше внимание, что замене подлежат только адрес хоста и порт; остальная часть URL остается неизменной.

В утилите два режима — выдача форматированного (по правилам SOAP, в виде XML) и неформатированного вывода. Первый показан на рис. 40.8. На нем вы, надеемся, без труда узнаете посылку клиента, вызывающего метод `echoString` с параметром `test`, и соответствующий ответ сервера.

Второй режим представляет собой дамп запроса HTTP и ответа сервера. Он может понадобиться в том случае, если клиент или сервер неправильно форматируют вывод, и он не может быть представлен в виде правильного XML-документа.

Резюме

Итак, мы закончили рассмотрение принципов функционирования Web-служб и инструментов для их создания и отладки. Вы познакомились с:

- стандартами мира Web-служб;
- вызываемыми (Invokable) интерфейсами и их регистрацией;
- компонентом `THTTPRio` и принципами настройки клиента на требуемую службу;
- созданием серверных приложений для Web-служб;
- пакетом MS SOAP Toolkit как альтернативным средством создания и использования служб;
- средством отладки для протокола SOAP — утилитой SOAP Trace.

Безусловно, Web-службы — это не единственная технология фирмы Borland, которая обеспечивает связь и устойчивую работу распределенных приложений. С другими технологиями вы познакомились в предыдущих главах и продолжите знакомиться в следующей. Однако именно Web-службы обещают стать универсальным средством для объединения разнородных платформ, поэтому мы рекомендуем вам обратить на них самое пристальное внимание.

Глава 41

Технология WebSnap

Для разработки распределенных Web-приложений, а точнее, их серверных частей (серверных Web-приложений), в Delphi существует еще одна технология — *WebSnap*. Являясь развитием более ранней технологии Web Broker *(см. главу 38)*, WebSnap оперирует примерно тем же набором программных инструментов, но более совершенна и позволяет решать гораздо более широкий круг задач. Вы увидите, что сложная архитектура приложения в WebSnap обеспечивает создание полнофункциональных серверных Web-приложений, а обширный набор компонентов позволяет делать это достаточно легко и непринужденно. Перечислим новые возможности разработки в WebSnap по сравнению с Web Broker.

Пожалуй главным и наиболее значимым нововведением стала возможность включать в шаблоны страниц скрипты, разработанные на языках VBScript или JScript. Но самое существенное — в этих скриптах разработчик может использовать набор типов и объектов, поддерживаемых WebSnap, позволяющих работать со свойствами и методами компонентов WebSnap прямо в страницах ответов.

Приложения WebSnap включают Web-страницы на основе шаблонов HTML или XML, что позволяет довольно существенно менять функциональность приложения без перекомпиляции исходного кода. Эта особенность может иметь решающее значение для систем, обслуживающих тысячи клиентов.

Возможность одновременного функционирования нескольких Web-модулей и компонентов-диспетчеров позволяет более эффективно управлять запросами, организуя параллельную обработку в нескольких нитях разнотипных запросов. А управление приложением в целом вынесено в специализированные компоненты и Web-модули.

В технологии WebSnap в вашем распоряжении имеется встроенный механизм регистрации и аутенификации пользователей.

Развитые средства генерации HTML-страниц позволяют не только создавать более совершенные страницы, но и просматривать промежуточные результаты прямо в процессе разработки.

Но при всем многообразии новых возможностей нельзя сказать, что технология WebSnap намного сложнее, чем Web Broker. Конечно же, на изучение новых компонентов уйдет время, но это время будет потрачено не зря. Итак, в этой главе рассматриваются следующие вопросы.

- Состав приложения WebSnap.
- Стандартные мастера WebSnap.
- Назначение модулей данных и модулей страниц.
- Роль компонента `TAdapter`.
- Диспетчеризация запросов клиентов.
- Разработка интерфейса и бизнес-логики приложения WebSnap.
- Использование шаблонов.
- Встроенные типы объектов и объекты.

Структура приложения WebSnap

Приложение WebSnap должно обеспечивать обработку запросов, поступающих от клиентов. Строго говоря, запросы принимаются Web-сервером и затем передаются приложению для обработки. При помощи специализированных компонентов (подробнее о них речь пойдет далее) приложение направляет запросы в хранилища бизнес-логики приложения, которые представляют собой модули данных и модули страниц HTML или XML.

Таким образом, приложение WebSnap состоит из двух основных частей (рис. 41.1).

Первая часть обеспечивает управление потоками запросов и ответов. Эта функциональность определяется компонентами WebSnap, которые умеют обрабатывать запросы и возвращать Web-серверу ответы, а также обеспечивают работоспособность страниц HTML, т. е. выполнение тегов HTML, скриптов VBScript или JScript и динамическую генерацию страниц на основе шаблонов HTML и XML.

Вторая часть приложения состоит из Web-модулей, обеспечивающих формирование интерфейса пользователя, доступ к источникам данных и хранение бизнес-логики приложения.

Рассмотрим перечисленные составные части подробнее.

Первоначально запрос поступает *компоненту-диспетчеру* `TPageDispatcher`, который "знает" о назначении *страниц* приложения и направляет им запрос для дальнейшей обработки и формирования ответа. При этом модуль страницы обеспечивает адекватную реакцию страницы на запрос.

При разработке приложения для каждой его страницы создается *шаблон*, который собственно и доступен для редактирования в среде разработки.

Шаблон создается на языках HTML или XML. А при необходимости реализовать более сложные функции разработчик может задействовать скрипты на языках VBScript или JScript.

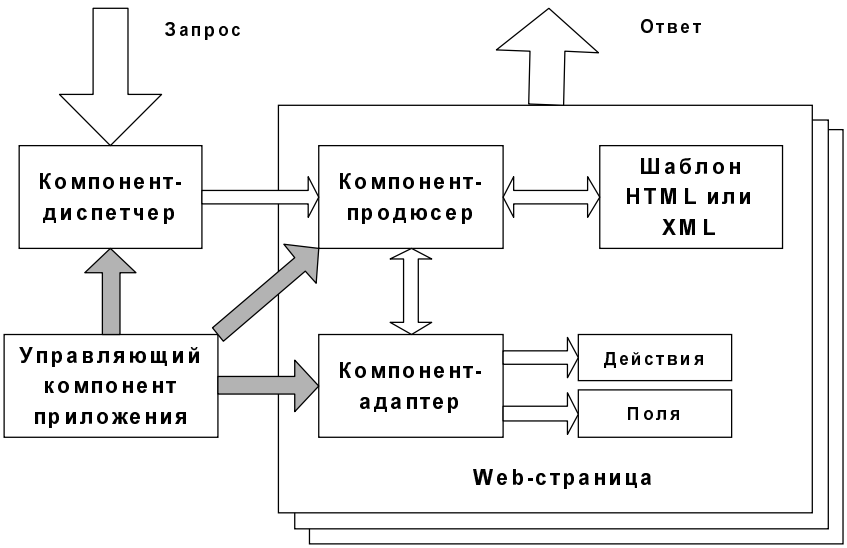


Рис. 41.1. Структура приложения WebSnap

На этапе выполнения приложения запрос поступает *компоненту-продюсеру*, связанному со страницей. Он обеспечивает генерацию страницы на основе имеющегося шаблона.

Примечание

Разнообразные компоненты-продюсеры доступны на странице **Internet** Палитры компонентов Delphi.

Если же шаблон содержит скрипт, то его выполнение обеспечивается *компонентами-адаптерами*. В зависимости от решаемых задач разработчик может выбрать несколько типов компонентов-адаптеров, доступных на странице **WebSnap** Палитры компонентов. Для направления запросов нужным адаптерам необходим компонент-диспетчер адаптеров `TAdapterDispatcher`.

Примечание

Трансляция скриптов выполняется автоматически специализированным процессором Script Engine.

Общая координация работы всех перечисленных компонентов и реализация бизнес-логики верхнего уровня осуществляются компонентом

`TWebAppComponents`. Он объединяет части приложения WebSnap в единую "технологическую цепочку", а разработчик может динамически заменять звенья цепи (перечисленные ранее компоненты) в зависимости от потребностей бизнес-логики приложения. Для этого предназначены свойства компонента, ссылающиеся на соответствующие компоненты адаптеры и диспетчеры.

Обработка запросов

Итак, давайте теперь проследим путь запроса в приложении WebSnap.

Первым делом запрос попадает к диспетчеру страниц, который решает, какая Web-страница будет отвечать на запрос. Например, если пользователь нажал некую кнопку на главной странице приложения, запрос будет направлен модулю главной страницы.

Здесь запрос обрабатывается компонентом-продюсером, который динамически изменяет страницу. Если же с нажатием кнопки связана какая-либо сложная операция, представленная скриптом, то выполнение запроса будет продолжено компонентом-диспетчером адаптеров. Он направит запрос соответствующему компоненту-адаптеру. Например, это может быть адаптер набора данных (компонент `TDataSetAdapter`), который, к примеру, обеспечит получение записей набора данных базы данных.

Web-модули

С точки зрения разработчика, приложение WebSnap в первую очередь состоит из специализированных модулей, на которых можно размещать необходимые компоненты. Эти Web-модули делятся на два вида: модули страниц и модули данных (рис. 41.2).

Модули страниц HTML реализуют для приложения его пользовательский интерфейс. Они выполняют две функции:

- содержат компонент-продюсер, обеспечивающий динамическую генерацию страницы на основе шаблона HTML или XML;
- содержат шаблон страницы.

Модули данных являются просто платформами для размещения любых невизуальных компонентов и аналогичны модулям данных, используемым в приложениях баз данных.

Кроме этого, Web-модули могут быть *модулями приложения*. Такие модули обеспечивают для приложения WebSnap хранилище компонентов, реализующих важнейшие функции для приложения в целом. Приложение может иметь только один такой Web-модуль, в качестве которого может выступать как модуль страницы, так и модуль данных.

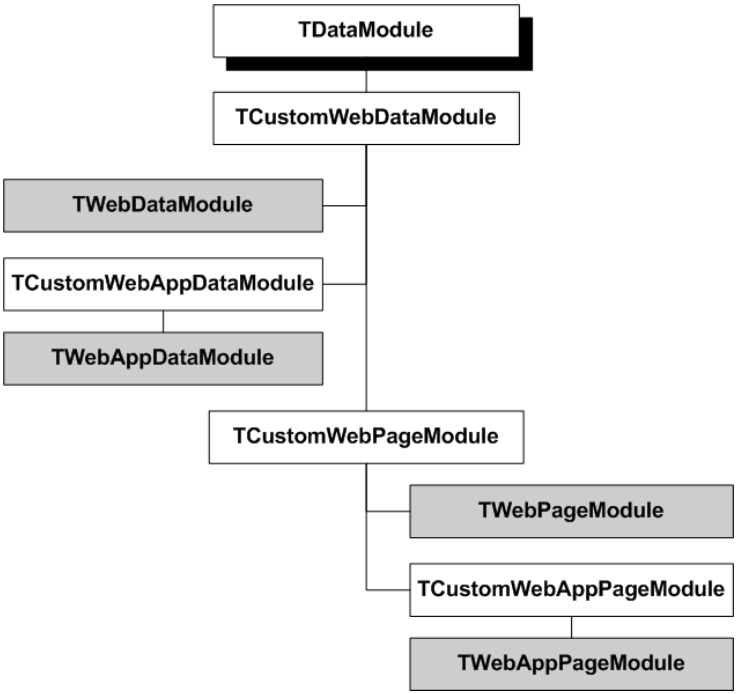


Рис. 41.2. Иерархия классов Web-модулей

Общим предком любого Web-модуля является класс `TDataModule`.

Модуль данных

Модуль данных приложения WebSnap (подобно обычному модулю данных) является платформой для любых невизуальных компонентов. На нем можно размещать как компоненты управления приложением WebSnap, так и, к примеру, компоненты доступа к данным.

Функциональность модуля данных инкапсулирована в классе `TWebDataModule`, который реализует не только контейнер для хранения невизуальных компонентов, но и обеспечивает управление запросами. Для управления запросами класс имеет два свойства.

Свойство

```
property Request: TWebRequest;
```

представляет запрос к приложению WebSnap. При помощи этого свойства разработчик получает доступ к многочисленным свойствам запроса.

Свойство

```
property Response: TWebResponse;
```

представляет собой ссылку на экземпляр класса ответа `TWebResponse`. Свойства и методы этого класса позволяют настроить и отправить ответ клиенту Web.

Также из модуля данных можно управлять сеансом доступа клиента. Для этого предназначено свойство

```
property Session: TAbstractWebSession;
```

А абстрактный класс сессии `TAbstractWebSession` позволяет реализовать для своих потомков методы установки длительности сеанса

```
property TimeoutMinutes: Integer
```

и прерывания работы

```
procedure Terminate;
```

Модуль страницы

Модуль страницы обеспечивает создание одной Web-страницы для приложения WebSnap. Помимо обычных для любой формы Delphi файлов, модуль страницы дополнительно имеет файл шаблона HTML (файл с расширением html) или XML (файл с расширением xml), который содержит описание страницы в соответствующем формате. Шаблон может содержать как обычные команды языка, так и скриптовые вставки на языках VBScript или JScript.

Редактор кода Delphi для модулей страниц обеспечивает отображение дополнительных файлов (рис. 41.3). Дополнительные закладки, расположенные справа внизу, позволяют просмотреть и редактировать не только шаблон HTML для страницы (вкладка **HTML Result**), но и просматривать ее внешний вид без компиляции приложения так, будто приложение выполняется (вкладка **Preview**). Эта функция позволяет очень быстро и легко создать и настроить пользовательский интерфейс страницы.

Рис. 41.3. Окно Редактора кода Delphi для модуля страницы

Класс `TWebPageModule` инкапсулирует функциональность Web-страницы.

На каждом модуле страницы должен быть размещен компонент-продюсер `TPageProducer`. Это обязательный компонент, т. к. он обеспечивает генерацию страницы из шаблона на этапе выполнения приложения. Для связывания компонента и модуля страницы предназначено свойство модуля

```
property PageProducer: IProduceContent;
```

которое должно ссылаться на компонент-продюсер. В общем случае страница приложения может иметь сложные функции, которые реализуются при помощи различных компонентов-продюсеров. Например, это может быть обычный компонент `TPageProducer` и компонент-продюсер `TDataSetPageProducer` для организации доступа к какой-либо базе данных. В этом случае разработчик может переназначать свойство `PageProducer` модуля страницы динамически во время выполнения.

А свойство

```
property DefaultAction: TAdapterActionName;
```

позволяет связать модуль страницы с одним из компонентов-адаптеров, которые разработчик может самостоятельно перенести в модуль из Палитры компонентов. Класс `TAdapterActionName` позволяет задать и компонент-адаптер, и его действие по умолчанию.

Дополнительно к уникальным класс модуля страницы имеет и обычные для Web-модуля свойства — ведь он также обеспечивает управление запросами клиентов. Поэтому здесь доступны свойства запроса и ответа Web:

```
property Request: TWebRequest;
property Response: TWebResponse;
```

Кроме этого, в модуле страницы можно управлять и сеансом соединения с клиентом при помощи свойства

```
property Session: TAbstractWebSession;
```

Модуль приложения

Базовая функциональность приложения WebSnap обеспечивается модулем приложения. Этот модуль должен содержать компоненты, инкапсулирующие функции общего управления приложением и обработку запросов.

Приложение WebSnap может иметь только один модуль приложения, хотя это может быть модуль страницы или модуль данных. Функциональность модуля приложения реализуют два класса.

Класс `TWebAppPageModule` представляет собой модуль страницы приложения. Он используется, если приложению необходима главная страница и с ее помощью удобно реализовать общее управление.

Класс `TWebAppDataModule` является модулем данных приложения. Он не является частью пользовательского интерфейса приложения и недоступен во время выполнения.

Любой модуль приложения наследует свойства и методы от соответствующего класса обычного модуля страницы или модуля данных (см. рис. 41.2) и обладает всеми их возможностями по обработке запросов и реализации действий.

Дополнительно модуль приложения имеет свойство

```
property AppServices: TComponent;
```

которое должно ссылаться на компонент `TWebAppComponents`. Этот компонент добавляется автоматически при создании модуля приложения и обеспечивает координацию работы остальных компонентов.

Дополнительно к компоненту `TWebAppComponents` по умолчанию при создании модуля приложения на нем размещаются компоненты-диспетчеры страниц и адаптеров.

Примечание

Если модуль приложения является модулем Web-страницы, то при его создании дополнительно создается и компонент `TPageProducer`, как для обычного модуля страницы.

Компоненты уровня приложения

Помимо специализированных компонентов WebSnap, в приложении доступны несколько дополнительных компонентов, которые обеспечивают выполнение функций на уровне приложения, а не отдельных запросов или страниц. Именно эти компоненты объединяют модули страниц и данных и обслуживающие их компоненты.

В технологии WebSnap компоненты уровня приложения выполняют следующие функции:

- управляют процессом взаимодействия компонентов WebSnap друг с другом и запросами;
- обеспечивают хранение информации о приложении в целом;
- обеспечивают работу приложения в многопользовательском режиме и позволяют вести списки пользователей;
- отслеживают взаимодействие приложения с файлами данных.

Управление компонентами приложения

Для того чтобы объединить все функциональные части приложения WebSnap в единое целое, применяется компонент `TWebAppComponents`. Он имеет ряд свойств, ссылающихся на другие нужные приложению компоненты (рис. 41.4). Разработчик имеет возможность изменять функциональность приложения, заменяя необходимые компоненты на этапе выполнения.

Компонент `TWebAppComponents` размещается в модуле приложения.

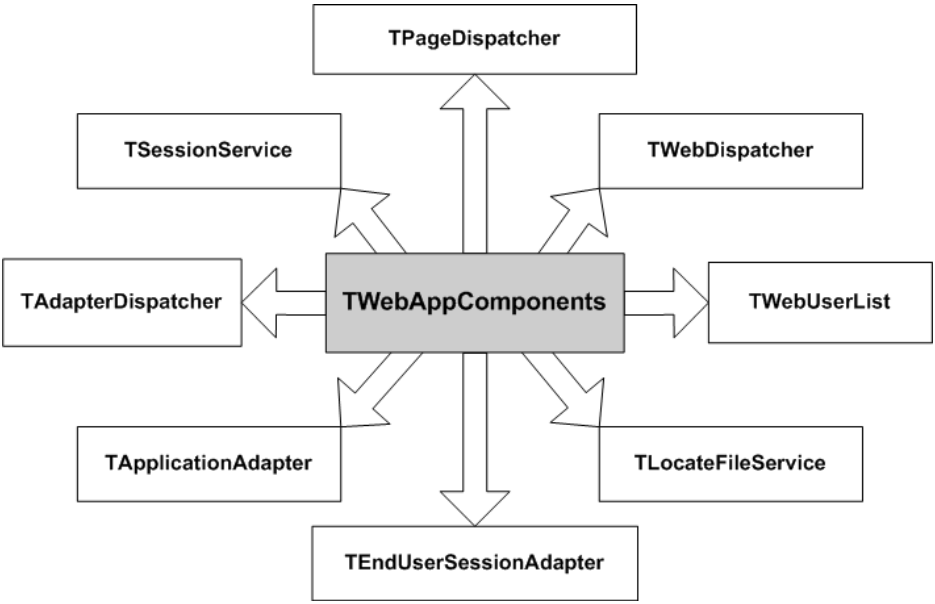


Рис. 41.4. Компонент `TWebAppComponents` управляет взаимодействием компонентов приложения WebSnap

Примечание

При переносе в модуль приложения первого компонента WebSnap из числа настраиваемых в компоненте `TWebAppComponents` его соответствующие свойства связываются с компонентами автоматически.

Свойство

```
property AdapterDispatcher: IAdapterDispatcher;
```

позволяет определить используемый приложением компонент `TAdapterDispatcher`.

Свойство

```
property ApplicationAdapter: IWebApplicationInfo;
```

ссылается на компонент `TApplicationAdapter`, который содержит общие для всего приложения параметры.

Свойство

```
property PageDispatcher: IPageDispatcher;
```

указывает на компонент `TPageDispatcher`, который будет управлять запросами и направлять их Web-страницам для обработки.

Свойство

```
property LocateFileService: ILocateFileService;
```

ссылается на компонент `TLocateFileService`, который предоставляет разработчику методы-обработчики событий, необходимые для управления файлами приложения. Это могут быть файлы данных, скриптов или шаблонов страниц. Методы-обработчики вызываются при обращении приложения к файлу.

Свойство

```
property Sessions: ISessionsService;
```

задает ссылку на компонент `TSessionsService`. Этот компонент управляет сессиями приложения, если с ним одновременно работают несколько пользователей.

Список всех пользователей приложения и их паролей может храниться в компоненте `TWebUserList`. А свойство

```
property UserListService: IWebUserList;
```

позволяет подключить его для использования приложением.

Параметры текущего пользователя приложения сохраняются в компоненте `TWebEndUser`, а сам он связывается с приложением свойством

```
property EndUserAdapter: IWebEndUser;
```

И наконец свойство

```
property DispatchActions: IWebDispatchActions;
```

связывает приложение с компонентом `TWebDispatcher`. Напомним, что этот компонент обеспечивает обработку Web-запросов приложения обычным модулем данных. Для этого компонент инкапсулирует список действий.

При возникновении исключительной ситуации уровня приложения компонент `TWebAppComponents` обрабатывает это событие, а разработчику доступен метод-обработчик

```
type
  TWebExceptionEvent = procedure (Sender: TObject; E: Exception;
  var Handled: Boolean) of object;
property OnException: TwebExceptionEvent;
```

Адаптер и глобальная переменная приложения

В приложении может быть собственный набор действий, не связанный с каким-либо из компонентов-адаптеров модулей страниц или данных. Для этого служит компонент `TApplicationAdapter`, который размещается в модуле приложения.

Так же, как и любой компонент-адаптер, он имеет свойство

```
property Actions: TAdapterDefaultActions;
```

для хранения набора действий и свойство

```
property Data: TAdapterDefaultFields;
```

для представления набора полей.

Отличие же заключается в том, что и поля, и действия компонента доступны из скриптов всего приложения при помощи глобальной переменной `Application`. Например, при создании модуля страницы приложения в его файл HTML автоматически добавляется следующая строка:

```
<h1><%= Application.Title %></h1>
```

которая при работе приложения обеспечит отображение общего заголовка приложения для каждой страницы.

В этом примере есть поле `Title`, которое создается для компонента по умолчанию и отображает значение опубликованного свойства

```
property ApplicationTitle: string;
```

компонента. В дальнейшей работе вы можете самостоятельно создать необходимые поля и действия и использовать их во всех страницах приложения.

Информация о пользователях

Приложение WebSnap может сохранять информацию о своих пользователях. Для этих целей имеется несколько компонентов.

Общий список возможных пользователей приложения формируется компонентом `TWebUserList`. Он обладает свойством

```
property UserItems: TWebUserItems;
```

которое представляет собой индексированный список объектов класса `TWebUserItem`. При этом каждый элемент списка соответствует одному пользователю.

Класс `TWebUserItem` позволяет задать для пользователя три свойства. Два из них имя и пароль:

```
property UserName: string
property Password: string
```

Третье свойство

```
property AccessRights: string
```

представляет собой список прав пользователя. Имена прав в свойстве должны быть разделены запятой, точкой с запятой или пробелом.

Если компонент `TWebUserList` связан с компонентом `TWebAppComponents` приложения, то при регистрации нового пользователя автоматически вызываются его методы-обработчики событий. При этом все они для идентификации пользователя используют параметр `UserID`. Значение идентификатора назначается пользователю автоматически.

Внимание

Методы-обработчики событий компонента `TWebUserList` не позволяют повлиять на решение об авторизации пользователя и служат только для программирования дополнительных операций. Подробнее о процессе авторизации пользователей см. далее в этой главе.

До и после авторизации пользователя вызываются методы-обработчики

```
type
  TValidateUserHandledEvent = procedure(Strings: TStrings; var UserID:
  Variant; var Handled: Boolean) of object;
property OnBeforeValidateUser: TValidateUserHandledEvent;
```

и

```
type
  TValidateUserEvent = procedure(Strings: TStrings; var UserID: Variant)
  of object;
property OnAfterValidateUser: TValidateUserEvent;
```

При проверке прав пользователя вызывается пара методов

```
type
  TCheckAccessRightsHandledEvent = procedure(UserID: Variant; Rights:
  string; var HasRight: Boolean; var Handled: Boolean) of object;
property OnBeforeCheckAccessRights: TCheckAccessRightsHandledEvent;
```

и

```
type
 TCheckAccessRightsEvent = procedure(UserID: Variant; Rights: string;
 var HasRight: Boolean) of object;
property OnAfterCheckAccessRights: TCheckAccessRightsEvent;
```

Если пользователь не найден в списке, вызывается метод-обработчик

```
property OnUserIDNotFound: TCheckAccessRightsHandledEvent;
```

А при неверно введенном имени пользователя или пароле вызывается метод-обработчик

```
type
  TValidateUserErrorEvent = procedure(Error: TValidateUserError; Strings:
  TStrings; var UserID: Variant; var Handled: Boolean) of object;
property OnValidateUserError: TValidateUserErrorEvent;
```

Обратите внимание, что здесь вы можете конкретизировать реакцию вашего приложения на ошибку с помощью параметра `TValidateUserError`:

```
TValidateUserError = (vuBlankPassword, vuBlankUserName,
vuUnknownUserName, vuUnknownPassword);
```

Например, в простейшем случае это можно сделать так:

```
procedure TSomePage.WebUserList1ValidateUserError(Error:
TValidateUserError; Strings: TStrings; var UserID: Variant;
var Handled: Boolean);
begin
 case Error of
  vuBlankPassword:   ShowMessage('Blank Password');
  vuBlankUserName:   ShowMessage('Blank User Name');
  vuUnknownPassword: ShowMessage('Wrong Password');
  vuUnknownUserName: ShowMessage('Wrong UserName');
 end;
end;
```

Информация о текущем пользователе во время выполнения приложения содержится в компоненте `TEndUserAdapter`. Для применения в скриптах страниц приложения компонент обеспечивает использование глобальной переменной `EndUser`.

Как и любой адаптер, компонент `TEndUserAdapter` содержит индексированные списки действий и полей в стандартных свойствах:

```
property Actions: TAdapterDefaultActions;
```

и

```
property Data: TAdapterDefaultFields;
```

И разработчик может определить для текущего пользователя все необходимые действия и переменные. Кроме этого, компонент содержит два предопределенных действия и два поля. Включая их в скрипты страниц, вы можете управлять доступом пользователя к ресурсам приложения.

Действие `LoginForm` позволяет управлять страницей регистрации пользователей. Для ее показа предусмотрен метод:

```
EndUser.LoginForm.Enabled
```

Указать имя страницы регистрации можно с помощью свойства

```
property LoginPage: string;
```

Действие `LogOut` отключает текущего пользователя.

Булево поле `LoggedIn` показывает, прошел пользователь аутенификацию, или нет. А поле `DisplayName` хранит имя пользователя.

Сессии

К приложению WebSnap одновременно могут обращаться несколько пользователей. Потому кроме стандартной задачи авторизации пользователей приложение должно уметь обрабатывать запросы от нескольких пользователей при произвольном порядке их поступления. Для этого предназначен компонент `TSessionsService`.

Для каждого нового подключившегося пользователя он создает собственный контекст, в котором сохраняются его специфические настройки. Этот контекст носит название *сессии* и имеет уникальный внутренний идентификатор. На этапе выполнения приложения список сессий компонент хранит в памяти.

Таким образом, каждый запрос пользователя идентифицируется с одной и только одной сессией, и на время обработки текущего запроса соответствующая сессия также становится текущей. При смене текущей сессии разработчику доступны два метода-обработчика:

```
type
 TStartSessionEvent = procedure(ASender: TObject; ASession:
 TAbstractWebSession) of object;
property OnStartSession: TStartSessionEvent;
```

и

```
type
  TEndSessionEvent = procedure(ASender: TObject; ASession:
  TAbstractWebSession; AReason: TEndSessionReason) of object;
property OnEndSession: TEndSessionEvent;
```

Как видно из приведенного объявления, текущая сессия может быть прервана приложением или по истечении срока ожидания:

```
type
  TEndSessionReason = (esTimeout, esTerminate);
```

А для всех сессий приложения компонент `TSessionsService` может задать время ожидания в неактивном состоянии. Для этого существует свойство

```
property DefaultTimeout: Integer;
```

Также можно задать максимальное число подключившихся пользователей, ограничив количество создаваемых сессий при помощи свойства

```
property MaxSessions: Integer;
```

Основываясь на информации о текущей сессии, компонент `TEndUserSessionAdapter` предоставляет сведения о текущем пользователе. Этот компонент подобен `TEndUserAdapter`, но обладает меньшим набором

методов-обработчиков событий. Дополнительно он позволяет получить идентификатор пользователя:

```
property UserID: Variant;
```

Управление файлами приложения

Приложение WebSnap активно использует вспомогательные файлы. Это шаблоны Web-страниц или файлы скриптов. Для контроля за использованием этих файлов применяется компонент `TLocateFileService`.

Если приложение пытается найти файл HTML, компонент вызывает метод-обработчик

```
type
  TLocateFileServiceFindFileEvent = procedure(ASender: TObject;
  AComponent: TComponent; const AFileName: string; var AFoundFile:
  string; var AHandled: Boolean) of object;
property OnFindTemplateFile: TLocateFileServiceFindFileEvent;
```

а если нужен файл скрипта, вызывается метод

```
property OnFindIncludeFile: TLocateFileServiceFindFileEvent;
```

Оба обработчика в качестве параметра `AFileName` получают полное имя файла и в случае успешного поиска возвращают имя найденного файла в параметре `AFoundFile`.

Компоненты-диспетчеры

Компоненты-диспетчеры играют важную роль в приложениях WebSnap. Они обеспечивают управление запросами клиентов в приложении. В этой категории технология WebSnap предоставляет два компонента.

Компонент `TPageDispatcher` направляет запросы соответствующим страницам приложения, при этом он не требует дополнительной настройки. При создании модуля приложения такой компонент размещается в нем и во время работы приложения автоматически выполняет свои функции.

Дополнительно разработчик может использовать методы-обработчики событий, которые возникают в процессе обработки запроса.

При поступлении запроса и после принятия решения о направлении запроса той или иной странице приложения вызываются методы-обработчики

```
type
  TDispatchPageHandledEvent = procedure(Sender: TObject; const PageName:
  string; var Handled: Boolean) of object;
property OnBeforeDispatchPage: TDispatchPageHandledEvent
```

и

```
type
  TDispatchPageEvent = procedure(Sender: TObject; const PageName: string)
  of object;
property OnAfterDispatchPage: TDispatchPageEvent
```

При этом идентификация страниц осуществляется по их именам, передаваемым параметром `PageName`.

При направлении запроса разработчик может управлять и доступом пользователей к странице. Для этого служит метод-обработчик

```
type
  TLoginRequiredEvent = procedure(Sender: TObject; const PageName:
string; var Required: Boolean) of object;
property OnIsLoginRequired: TLoginRequiredEvent;
```

Если авторизация пользователя необходима, параметр `Required` должен иметь значение `True`:

```
procedure TPageProducerPage2.PageDispatcherIsLoginRequired(Sender:
TObject; const PageName: String; var Required: Boolean);
begin
  Required := PageName = 'SecuredPage1';
end;
```

Если попытка регистрации пользователя была неудачной, вызывается метод-обработчик

```
type
 TPageAccessDeniedEvent = procedure(Sender: TObject; const PageName:
 string; Reason: TPageAccessDenied; var Handled: Boolean) of object;
property OnPageAccessDenied: TPageAccessDeniedEvent;
```

Если компонент-диспетчер не смог найти страницу — адресат запроса, вызывается метод-обработчик

```
property OnPageNotFound: TDispatchPageHandledEvent;
```

Компонент `TAdapterDispatcher` связывет события пользовательского интерфейса страниц приложения с некоторым компонентом-адаптером, обеспечивающим выполнение сложных операций на основе скриптов VBScript или JScript. Компонент также работает без дополнительной настройки. Его методы-обработчики позволяют разработчику отслеживать вызов полей или действий компонентов-адаптеров.

Компоненты-продюсеры

Компоненты-продюсеры являются обязательными элементами приложений WebSnap. Их задача — генерация HTML страниц на основе шаблонов HTML или XML во время выполнения. Семейство компонентов-продюсеров расположено на странице **Internet** Палитры компонентов.

Компонент-продюсер должен обязательно присутствовать во всех модулях страниц, используемых приложением WebSnap. По умолчанию в приложениях WebSnap при создании новых модулей страниц устанавливается компонент `TPageProducer`. Он предназначен для работы с шаблонами HTML.

Шаблон HTML задается свойством

```
property HTMLDoc: TStrings;
```

которое определяет сам шаблон, или свойством

```
property HTMLFile: TFileName;
```

которое задает имя файла, содержащего шаблон.

Если шаблон использует скрипты, свойство

```
property ScriptEngine: string;
```

позволяет определить язык скрипта.

На этапе выполнения приложения разработчик имеет возможность генерировать страницы по мере необходимости. Для этого достаточен метод

```
function Content: string;
```

но при этом обязательно должен быть задан шаблон страницы при помощи свойств `HTMLDoc` или `HTMLFile`. В противном случае можно применить два метода для генерации содержимого страницы "на лету".

Метод

```
function ContentFromString(const S: string): string;
```

работает с шаблоном, заданным параметром `S`.

Метод

```
function ContentFromStream(Stream: TStream): string;
```

сначала загружает шаблон из потока `Stream`.

Кроме компонента `TPageProducer` на странице **Internet** доступны еще четыре специализированных компонента-продюсера, которые обеспечивают загрузку данных для генерируемой страницы из связанного набора данных. Подробнее об этих компонентах *см. главу 38*.

Компоненты-адаптеры

Компоненты-адаптеры обеспечивают для приложений WebSnap удобный способ реализации их бизнес-логики. Адаптеры могут хранить наборы настроенных *действий* и *полей*.

Действия представляют собой процедуры, реализованные для данного адаптера, которые могут быть вызваны приложением при возникновении определенных событий. Например, при нажатии кнопки исполняется код соответствующего скрипта страницы и затем компонент `TAdapterProducer` обеспечивает выполнение действия компонента-адаптера (рис. 41.5).

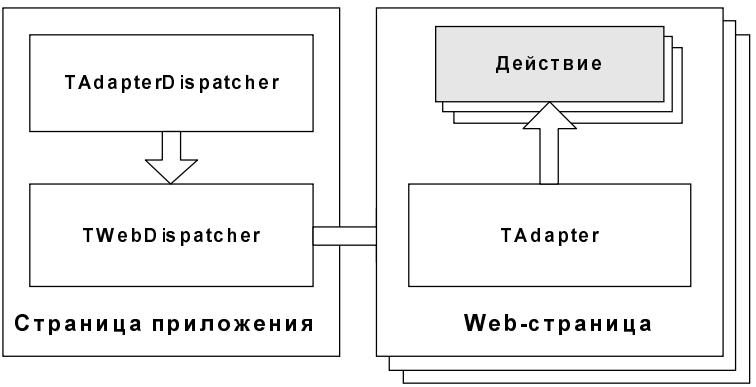


Рис. 41.5. Механизм вызова действий компонентов-адаптеров

Поля компонента-адаптера обеспечивают хранение переменных, необходимых при работе страниц приложений и выполнении действий.

Наиболее универсальным компонентом-адаптером из имеющихся в распоряжении технологии WebSnap является `TAdapter`.

Индексированный список действий адаптера инкапсулирован свойством

```
property Actions: TAdapterActions;
```

При щелчке на кнопке свойства в Инспекторе объектов появляется окно Редактора действий (рис. 41.6). Здесь вы можете создать или удалить объекты действий и настроить их. Отдельное действие представлено классом `TAdapterAction`.

При реализации действий главную роль играет метод-обработчик

```
type
 TActionExecuteEvent = procedure(Sender: TObject; Params: TStrings) of
 object;
property OnExecute: TActionExecuteEvent;
```

который активизируется при выполнении действия из скрипта страницы приложении. Здесь разработчик может предусмотреть выполнение любых необходимых операций. Параметр `Params` задает необходимые переменные действий.

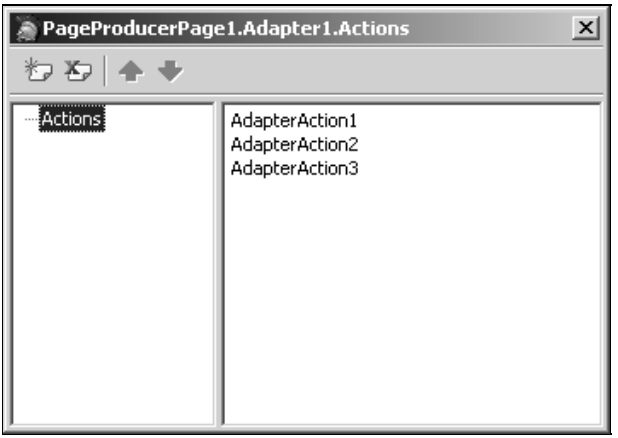


Рис. 41.6. Редактор действий компонента-адаптера

Если выполнение действия требует передачи параметров, доступ к ним можно получить в методе-обработчике события получения действием параметров

```
type
  TActionGetParamsEvent = procedure(Sender: TObject; Params: TStrings) of
  object;
property OnGetParams: TActionGetParamsEvent;
```

Поля компонентов-адаптеров позволяют хранить значения необходимых переменных. Индексированный список объектов полей адаптера хранится в свойстве

```
property Data: TAdapterFields;
```

При щелчке на кнопке свойства в Инспекторе объектов появляется Редактор, аналогичный Редактору действий (см. рис. 41.6). При создании нового объекта поля разработчик имеет возможность выбрать тип данных поля из списка. Описание возможных типов полей представлено в табл. 41.1.

Отдельное поле описывается базовым классом `TAdapterField`. Другие типы полей, представленные в табл. 41.1, имеют собственные классы, которые наследуют основные возможности от базового класса. Их основное отли-

чие — реализация методов-обработчиков для получения и задания значений в зависимости от поддерживаемых типов данных.

Таблица 41.1. *Типы полей для компонента-адаптера* `TAdapter`

Класс	Описание
`TAdapterField`	Универсальное поле для представления простых и строковых типов данных
`TAdapterBooleanField`	Специализированное поле для хранения булевых переменных
`TAdapterFileField`	Поле для хранения имен файлов, использованных при работе приложения WebSnap
`TAdapterImageField`	Поле для представления бинарных массивов
`TAdapterMemoField`	Поле для представления данных типа `Memo`
`TAdapterMultiValueField`	Поле используется при необходимости хранить несколько значений одновременно или для хранения одномерных массивов. Для доступа к отдельным значениям используется параметр `Index`

Поэтому рассмотрим класс `TAdapterField` более подробно.

При изменении значения поля вызывается метод-обработчик

```
type
  TAdapterFieldValueEvent = procedure (Sender: TObject; Value: Variant)
  of object;
property OnUpdateValue: TAdapterFieldValueEvent;
```

Для того чтобы значение поля было доступно из скрипта, требуется задать его в методе-обработчике

```
type
  TAdapterFieldGetValueEvent = procedure (Sender: TObject; var Value:
  Variant) of object;
property OnGetValue: TAdapterFieldGetValueEvent;
```

Обратите внимание, что работа с этими методами и со свойствами в классах Delphi аналогична. Простейший пример:

```
procedure TPageProducerPage2.AdapterField1GetValue(Sender: TObject; var
Value: Variant);
begin
 Value := FSomeInternalValue;
end;
```

Кроме этого, для каждого поля можно задать целый список возможных значений, который может применяться при создании элементов управления страниц (например, списков или радиогрупп). Для этого существует специализированный компонент-адаптер `TStringsValuesList`.

Он имеет свойство

```
property Strings: TStrings;
```

в котором в соответствии с форматом `Name=Value` можно задать необходимые значения.

В компоненте же `TAdapter` ссылка на компонент со списком значений задается в свойстве

```
property ValuesList: TComponent;
```

Более подробно о компонентах-адаптерах в приложениях WebSnap рассказывается далее в этой главе.

Создание приложения WebSnap

Для создания приложения WebSnap и его составных частей — модулей страниц и данных в состав Delphi входят специализированные мастера. Они доступны в Репозитории Delphi на странице **WebSnap**.

Мастер создания приложения вызывается при щелчке на значке **WebSnap Application**. Его окно представлено на рис. 41.7.

Группа радиокнопок **Server Type** предназначена для выбора типа создаваемого приложения (типы приложений совпадают для технологий WebSnap и Web Broker).

- **ISAPI/NSAPI Dynamic Link Library** — динамическая библиотека, работающая в адресном пространстве сервера Internet Information Server в многопользовательском режиме.
- **CGI Stand-alone executable** — исполняемый файл CGI (Common Gateway Interface) получающий управляющие команды через переменные окружения.
- **Web App Debugger executable** — исполняемый файл, позволяющий отлаживать приложение средствами Delphi в отсутствие реального Web-сервера.

В процессе своей работы мастер создаст главную страницу приложения. В зависимости от того, что будет выбрано в группе радиокнопок **Application Module Components**, в качестве модуля приложения будет использована страница HTML или модуль данных.

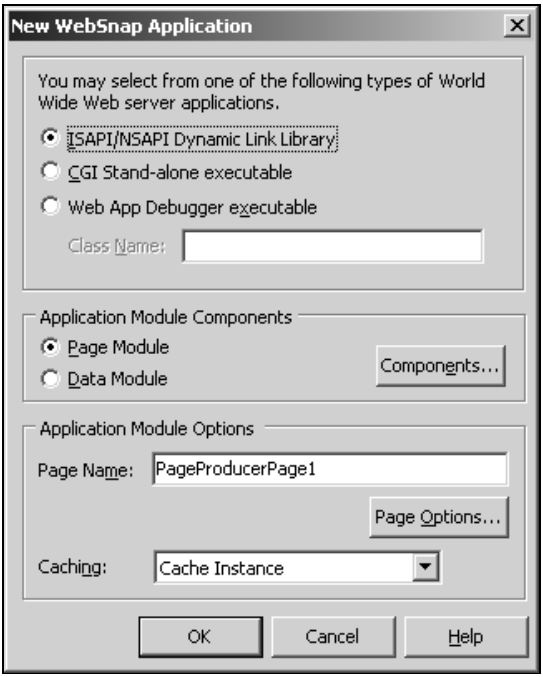


Рис. 41.7. Мастер создания приложения WebSnap

Здесь же располагается кнопка **Components**, при нажатии на которую разворачивается окно **Web App Components** (рис. 41.8). В нем можно выбрать компоненты WebSnap, которые будут размещены в модуле приложения. По умолчанию выделены три компонента: `TApplicationAdapter`, `TPageDispatcher` и `TAdapterDispatcher`. Их назначение вам уже известно.

Примечание

Кроме перечисленных, мастер включает в приложение в обязательном порядке компонент `TWebAppComponents`.

Затем в строке ввода **Page Name** задать имя создаваемой страницы взамен предложенного по умолчанию.

Список **Caching** позволяет настроить поведение создаваемого модуля после обработки запроса клиента на этапе выполнения приложения. Если модуль не выполняет важных функций или используется редко, можно задать значение **Destroy Instance**. Тогда сразу после выполнения запроса модуль удаляется из памяти. Значение **Cache Instance** позволяет сохранить модуль в памяти.

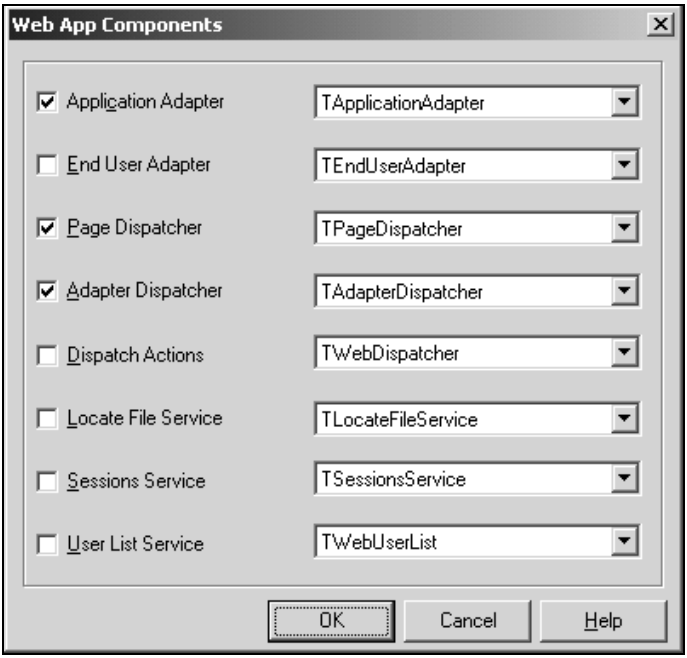


Рис. 41.8. Окно **Web App Components** выбора компонентов для Web-модуля

Для модуля приложения рекомендуется устанавливать значение **Cache Instance**.

При щелчке на кнопке **Page Options** разворачивается окно **Application Module Page Options**, в котором можно настроить дополнительные параметры Web-модуля (рис. 41.9).

Здесь в списке **Type** группы **Producer** можно выбрать компонент-продюсер, который будет обеспечивать генерацию страницы во время выполнения приложения. Если для создаваемой страницы планируются скрипты, то в списке **Script Engine** нужно выбрать поддерживаемый язык скрипта.

Нижеследующая группа меняет свое название в зависимости от выбранного типа компонента-продюсера. Если это `TXSLPageProducer`, группа называется **XSL**, иначе — **HTML**. И в ней задаются параметры соответствующего шаблона.

Включенный флажок **New File** обеспечивает создание для шаблона отдельного файла. Его имя совпадает с именем модуля исходного кода Object Pascal, а расширение зависит от выбора компонента-продюсера (HTML или XSL).

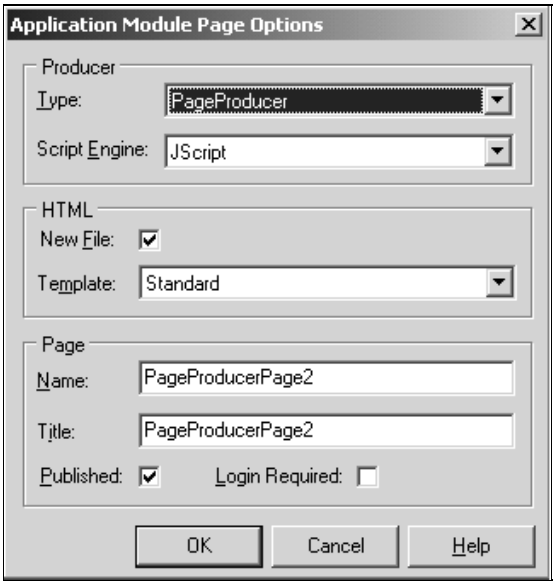


Рис. 41.9. Окно **Application Module Page Options** служит для дополнительной настройки создаваемого Web-модуля

Затем в списке **Template** нужно задать тип шаблона. По умолчанию установлено значение **Standard** и файл шаблона будет заполнен в соответствии с выбранным языком скрипта. Значение **Blank** запрещает генерацию исходного шаблона. Значение **Data Packet** необходимо только для шаблонов XSL и генерирует соответствующий шаблон.

В строках **Name** и **Title** задаются имя и заголовок страницы соответственно.

Включенный флажок **Published** позволяет странице автоматически отвечать на запрос, если запрос адресован странице напрямую.

Флажок **Login Required** включает авторизацию пользователей при доступе к странице.

После настройки всех параметров будущего приложения и нажатия кнопки **OK** мастера будут созданы исходные файлы нового проекта Delphi и модуля приложения.

Доступ к файлам модуля приложения (PAS, HTML, XML или XSL) и любых других Web-модулей обеспечивают закладки в правой нижней части Редактора Delphi.

При необходимости добавить к приложению новые страницы или модули данных также применяются мастера Репозитария Delphi.

Элементы управления мастера создания страницы уже были описаны ранее. Единственный дополнительный список **Creation** позволяет выбрать способ создания страницы. При значении **On Demand** страница будет создана только при первом обращении к ней. Значение **Always** задает обязательное создание страницы.

При создании модуля данных необходимо задать только способы создания и кэширования.

Конструирование интерфейса и управление данными

Теперь, имея представление о структуре приложения WebSnap, перейдем к небольшому практикуму и рассмотрим некоторые особенности использования компонентов WebSnap.

Воспользовавшись мастером создания приложения WebSnap, создадим новое приложение. В качестве типа сервера (см. рис. 41.7) выберем Web App Debugger executable. В этом случае исполняемый файл приложения обеспечит отображение Web-страниц приложения средствами Delphi. Зададим имя страницы — `MainPage`. Все остальные параметры оставим без изменений.

Выберем в модуле страницы приложения компонент `TApplicationAdapter` и зададим для его свойства `ApplicationTitle` значение `Simple WebSnap Application`. Обратите внимание, что в дальнейшем при добавлении к приложению новых страниц этот общий заголовок будет отображаться на них за счет того, что в шаблоны страниц будет автоматически добавлен тег

```
<h1><%= Application.Title %></h1>
```

где `Application` — глобальная переменная приложения, предоставляемая компонентом `TApplicationAdapter`.

После сохранения и компиляции проекта приложение доступно для выполнения и отладки. Для этого необходимо запустить утилиту WebApp Debugger (рис. 41.10), выбрав одноименный пункт из меню **Tools** Delphi.

Если сервер еще не запущен, это необходимо сделать при помощи кнопки **Start**. После этого выберите ссылку **Default URL** и затем в развернувшемся окне Internet Explorer — ссылку **ViewDetails**. Из появившегося списка выберите и запустите тестовое приложение — появится главная страница приложения MainPage.

По аналогичной схеме мы будем тестировать все последующие этапы разработки.

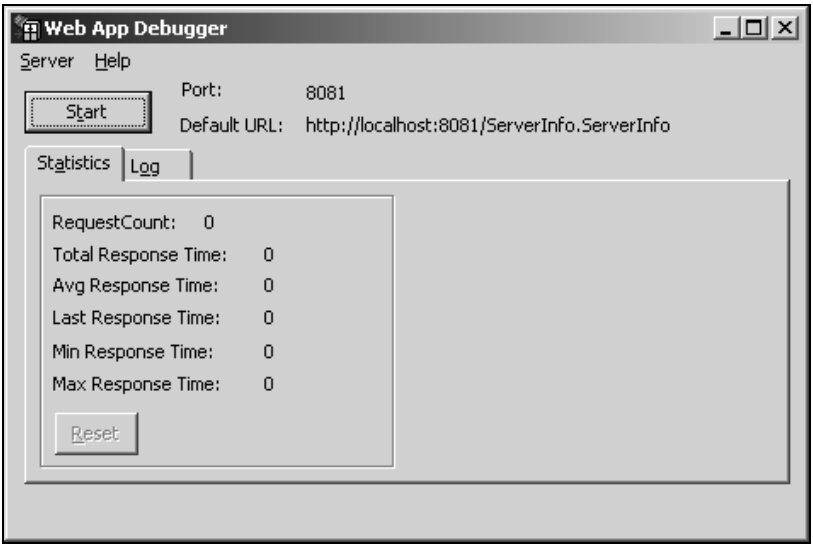


Рис. 41.10. Утилита Web App Debugger для отладки приложений WebSnap

Вопросы создания пользовательского интерфейса на основе Web-страниц выходят за рамки нашего обсуждения. Заинтересованные читатели могут обратиться к богатейшему выбору литературы по этой теме. Здесь же отметим, что в окне Редактора Delphi вы также можете работать с текстом шаблона напрямую. При необходимости использовать профессиональный инструментарий можно просто заменить HTML или XML-файлы страниц приложений в папке проекта на созданные в специализированных редакторах, сохраняя, естественно, имена файлов.

Использование скриптов

В шаблонах страниц ответов в технологии WebSnap допустимы скрипты, написанные на языках JScript или VBScript. Языком по умолчанию считается JScript. Обратите внимание, что эти скрипты не пересылаются клиенту для реализации той или иной функции пользовательского интерфейса, а исполняются на стороне сервера при динамической генерации страницы ответа. К примеру, это могут быть код, формирующий общий заголовок для всех страниц приложения, автоматическая нумерация и именование страниц и т. д.

Фрагмент, написанный на одном из этих языков, включается в состав страницы HTML между тегами `<% %>`.

Помимо стандартных конструкций языков JScript или VBScript, есть ряд объектов, создание которых во время выполнения берет на себя приложе-

ние. Эти объекты есть ни что иное, как представление в скрипте компонентов приложения, их свойств и методов. Их структура повторяет структуру приложения WebSnap.

Например, чуть ранее вы уже встретились с глобальным объектом приложения `Application`:

```
<h1><%= Application.Title %></h1>
```

Он обеспечивает доступ к общим свойствам приложения.

Объект `Modules` представляет собой перечисление (enumeration) объектов модулей, содержащихся в приложении. А объект `Pages` аналогичным образом содержит список объектов `Page` для каждой страницы приложения.

Другие объекты позволяют работать со свойствами сессии, текущего пользователя, продюсера, запросов и ответов.

Навигация по страницам приложения

Теперь добавим к проекту новую страницу при помощи мастера из Репозитория Delphi. Сам процесс не представляет сложности. Однако если приложение WebSnap имеет больше одной страницы, в шаблоне главной страницы начинает работать код, обеспечивающий навигацию по страницам приложения (листинг 41.1).

Листинг 41.1. Фрагмент кода шаблона главной страницы HTML приложения, обеспечивающий навигацию по страницам

```
<table cellspacing="0" cellpadding="0">
<td>
<%  e = new Enumerator(Pages)
     s = ''
     c = 0
     for (; !e.atEnd(); e.moveNext())
     {
       if (e.item().Published)
       {
         if (c>0) s += ' | '
         if (Page.Name != e.item().Name)
           s += '<a href="' + e.item().HREF + '">' + e.item().Title +
           '</a>'
         else
           s += e.item().Title
```

```
          c++
        }
      }
      if (c>1) Response.Write(s)
%>
</td>
</table>
```

Этот код встраивается в шаблон главной страницы приложения при ее создании.

Здесь создается новый перечислитель, т. е. содержащий страницы приложения, доступные для просмотра. Затем в цикле в строку `s` добавляются имена страниц и, если счетчик страниц содержит больше одной страницы, панель навигации по страницам отображается в главной странице приложения. Это осуществляется методом `Response.Write(s)`, где `Response` — ссылка на объект ответа на запрос.

Использование полей и действий компонентов-адаптеров

Компоненты-адаптеры WebSnap предоставляют разработчику два важнейших инструмента, необходимых для разработки интерактивных пользовательских интерфейсов. Это индексированный список полей, которые обеспечивают хранение переменных различных типов во время выполнения приложения и список действий, позволяющих реализовывать функции интерфейса приложения.

В качестве примера попробуем создать простую страницу HTML, обеспечивающую просмотр текстовых файлов. Для этого к созданному ранее приложению добавим новую страницу `TextFileViewer`.

Шаблон для этой страницы создан при помощи Microsoft Front Page и содержит однострочный редактор для ввода имени файла и многострочный редактор для просмотра содержимого файла. Шаблон был сохранен под именем uTextFileViewer.html и перенесен в папку проекта. При этом описание заголовка страницы было изменено — скопировано из исходного файла HTML, сгенерированного Delphi при создании файла uTextFileViewer.html.

Затем в Delphi для компонента-адаптера создаваемой страницы создадим новое поле. Для этого необходимо щелкнуть на кнопке свойства `Data` в Инспекторе объектов и в появившемся Редакторе полей создать новое поле. Тип поля — `TAdapterFileField`. В этом поле будет храниться текущее имя файла. Затем для представления выбранного файла создадим еще одно поле, выбрав для него тип `TAdapterMemoField`.

Для новых полей создадим методы-обработчики события `OnGetValue`, как это показано в листинге 41.2.

Поле `FileName` возвращает полное имя текущего файла, а `FileContent` возвращает содержимое этого файла в формате Memo. При этом его метод-обработчик `FileContentGetValue` через значение переменной `FFileName` создает объект файлового потока и заполняет переменную `FFileContent`.

Теперь необходимо создать действие, обеспечивающее выбор файла и его отображение на странице после нажатия кнопки **Open**. Для этого откроем Редактор действий компонента-адаптера, нажав кнопку свойства `Actions` в Инспекторе объектов (см. рис. 41.6). Создав в нем новый объект класса `TAdapterAction`, назовем его `SelectFileName`. Затем создадим метод-обработчик `OnExecute` и поместим в него код для вызова стандартного диалога открытия файла (листинг 41.2).

Листинг 41.2. Исходный код модуля страницы `TextFileViewer`

```
implementation

{$R *.dfm}  {*.html}
uses WebReq, WebCntxt, WebFact, Variants;
function TextFileViewer: TTextFileViewer;
begin
  Result := TTextFileViewer(WebContext.FindModuleClass(TTextFileViewer));
end;
procedure TTextFileViewer.FileNameGetValue(Sender: TObject;  var Value:
Variant);
begin
 Value := FFileName;
end;
procedure TTextFileViewer.FileContentGetValue(Sender: TObject; var Value:
String);
begin
 if FFileName = ''
  then FFileContent := ''
  else
  try
   with TFileStream.Create(FFileName, fmOpenRead) do
   try
    SetString(FFileContent, nil, Size);
    Read(Pointer(FFileContent)^, Size);
```

```
   finally
    Free;
   end;
  except
   on E: Exception do FFileContent := E.Message;
  end;
 Value := FFileContent;
end;
procedure TTextFileViewer.SelectFileNameExecute(Sender: TObject; Params:
TStrings);
begin
 if dlgOpen.Execute
  then FFileName := dlgOpen.FileName;
end;
initialization
  if WebRequestHandler <> nil then
WebRequestHandler.AddWebModuleFactory(TWebPageModuleFactory.Create(TTextF
ileViewer, TWebPageInfo.Create([wpPublished {, wpLoginRequired}],
'.html'), crOnDemand, caCache));
end.
```

Теперь необходимо связать созданные поля и действия с элементами управления страницы uTextFileViewer.html. Поле `FileName` должно отображаться в однострочном текстовом редакторе `edFileName`. Для этого объявление элемента управления в шаблоне страницы должно выглядеть так:

```
<input type="text" name="edFileName" size="38"
value=<%=FileViewerAdapter.FileName.Value%>>
```

Поле `FileContent` должно отображаться в многострочном редакторе `meFileContent`. Его объявление должно выглядеть так:

```
<textarea rows="19" name="meFileContent" cols="82">
  <%=FileViewerAdapter.FileContent.Value%>
</textarea>
```

При нажатии кнопки `buOpen` должен вызываться метод-обработчик `OnExecute` действия `SelectFileName`. Для этого шаблон страницы должен содержать следующие объявления:

```
<input type=hidden name="__action"
value="<%=FileViewerAdapter.SelectFileName.AsFieldValue%>">
<input type="submit" value="Open" name="buOpen" on-
click="F.__action.value=
'<%=FileViewerAdapter.SelectFileName.AsFieldValue%>'">
```

Здесь событие нажатия кнопки связывается с выполнением действия `SelectFileName`, т. е. с выполнением его метода обработчика `OnExecute`.

Все действия наследуют свойства общего интерфейса `IAdapterActionWrapper`. И свойство

```
property AsFieldValue: WideString
```

возвращает результат выполнения действия в строковом виде.

В результате после компиляции и запуска приложения страница `TextFileViewer` позволяет при нажатии кнопки **Open** выбирать из стандартного диалога файлы и просматривать их в многострочном редакторе. При этом имя файла отображается в однострочном редакторе.

Взаимодействие с базами данных

Теперь остановимся на вопросах взаимодействия приложений WebSnap с базами данных. Рассмотрим пример, в котором данные из таблицы Country демонстрационной базы данных Delphi отображаются на странице HTML.

Для этого к рассмотренному ранее приложению SimpleWebSnap Application добавим новую страницу и назовем ее DataViewer. Однако в этом случае будут использованы другие компонент-адаптер и компонент-продюсер. Так как в простейшем случае удобнее всего отображать данные в табличном виде, вместо стандартного компонента-продюсера `TPageProducer` выберем компонент `TAdapterPageProducer`, а вместо славно потрудившегося в предыдущих примерах компонента `TAdapter` — `TDataSetAdapter`. И теперь давайте посмотрим, какие же преимущества дают новые компоненты.

Специализированные элементы управления

Компонент `TAdapterPageProducer` предоставляет дополнительные возможности по созданию Web-страниц и связыванию элементов управления этих страниц с полями и действиями компонента-адаптера. Для этого необходимо нажать кнопку в строке свойства

```
property WebPageItems: TWebComponentList;
```

компонента в Инспекторе объектов. Появится окно редактора свойства, представленное на рис. 41.11.

В верхней части окна расположен иерархический список, подобный редакторам полей и действий компонентов-адаптеров (см. рис. 41.6). Список предназначен для создания, редактирования и удаления элементов управления Web-страницы. В нижней части окна расположен многостраничный блокнот, визуализирующий текст шаблона и внешний вид страницы.

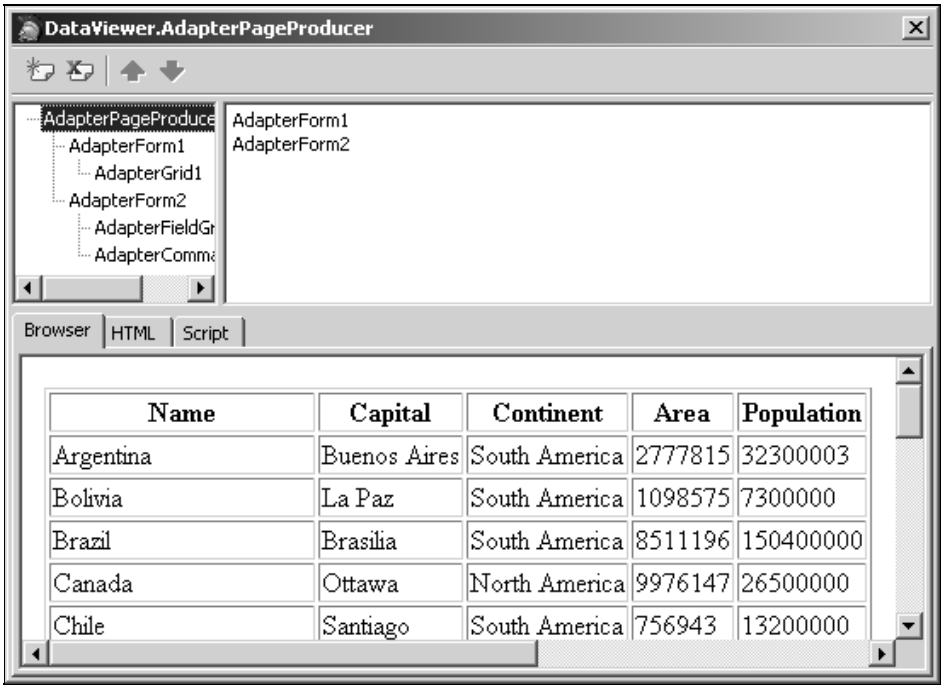


Рис. 41.11. Окно редактора компонента `TAdapterPageProducer`

Редактор позволяет создавать пользовательский интерфейс страницы со многими уровнями вложенности. В корне дерева (его имя совпадает с именем компонента) элементов управления доступны только два вида объектов.

Класс `TAdapterForm` инкапсулирует возможности формы HTML. На нем можно размещать элементы управления.

Класс `TLayoutGroup` обеспечивает выравнивание элементов управления в нескольких колонках. На экземпляре этого класса можно помещать только формы HTML. Если свойство класса `TLayoutGroup`

```
property DisplayColumns: Integer;
```

имеет значение −1 (значение по умолчанию) все дочерние формы (экземпляры класса `TAdapterForm`) размещаются на странице без выравнивания — одна под другой. В противном случае значение свойства определяет число колонок, в которых будут помещены формы на странице слева направо.

В форме HTML для компонента `TAdapterPageProducer` доступны следующие элементы управления.

Класс `TAdapterFieldGroup` обеспечивает представление полей связанного набора данных в виде текстового списка имен и значений полей. Одновре-

менно отображаются значения полей только одной записи. Для соединения с набором данных предназначено свойство

```
property Adapter: TComponent;
```

которое указывает на компонент `TDataSetAdapter` и через него на компонент набора данных.

Этот класс всегда используется совместно с классом `TAdapterCommandGroup`, который представляет собой набор кнопок, обеспечивающих навигацию по набору данных. Его свойство

```
property DisplayComponent: TComponent
```

должно ссылаться на экземпляр класса `TAdapterFieldGroup`, расположенный в той же форме.

Класс `TAdapterGrid` инкапсулирует таблицу. Для ее заполнения данными достаточно указать на нужный компонент `TDataSetAdapter` также при помощи свойства

```
property Adapter: TComponent;
```

Дополнительно к этому свойства

```
property TableAttributes: TGridAttributes;
```

и

```
property RowAttributes: TGridRowAttributes;
```

позволяют соответственно настраивать внешний вид таблицы и ее строк.

Класс `TAdapterErrorList` не является визуальным элементом управления, поэтому он недоступен в иерархическом дереве редактора и отображается после создания только в списке в правой части редактора. Он предназначен для обработки возможных исключительных ситуаций с наборами данных.

Просмотр данных

Для нашего примера будет достаточно отобразить данные в табличной форме. Для этого в редакторе компонента `TAdapterPageProducer` добавим форму HTML (экземпляр класса `TAdapterForm`) и в ней — таблицу (экземпляр класса `TAdapterGrid`). Результат наших несложных приготовлений можно увидеть здесь же. Для этого достаточно выбрать закладку **Browser** в нижней части окна редактора, текст шаблона доступен в закладке **Script**. Однако на этом этапе мы не видим ни данных, ни таблицы. Вместо этого отображается сообщение об ошибке (см. рис. 42.12).

Тем не менее все идет по плану. Дело в том, что мы не связали табличный элемент управления с набором данных.

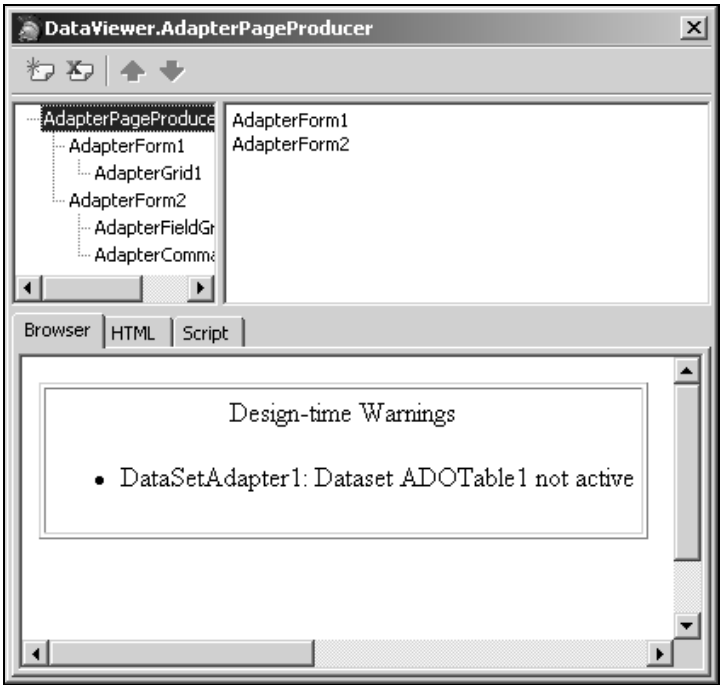


Рис. 42.12. Сообщение об ошибке в закладке **Browser** редактора компонента `TAdapterPageProducer`

Примечание

В компоненте `TAdapterPageProducer` каждый из созданных элементов управления (если, конечно, он предназначен для этого) может быть связан с собственным набором данных. Это позволяет отображать на одной странице данные из нескольких источников. Очевидно, что данный подход более гибок, чем назначение набора данных всему компоненту `TAdapterPageProducer`.

Это делается не напрямую, а через посредника — компонент-адаптер. Элементы управления компонента `TAdapterPageProducer` имеют свойство `Adapter`, которое позволяет связать их с компонентом-адаптером. В нашем случае это должен быть компонент `TDataSetAdapter`, т. к. он "умеет" взаимодействовать с компонентами наборов данных. Для этого предназначено его свойство

```
property DataSet: TDataSet;
```

И поскольку оно имеет тип `TDataSet`, адаптер может работать с компонентами различных технологий доступа к данным. В нашем примере выберем компоненты ADO. Для этого поместим в модуль создаваемой страницы

компоненты TADOConnection и TADOTable и настроим их для доступа к таблице Country базы данных Microsoft Access DBDEMOS.MDB, которая входит в состав демонстрационной базы данных Delphi и по умолчанию располагается в папке ..\\Program Files\Common Files\Borland Shared\Data. *(Подробнее о настройке этих компонентов см. главу 23.)*

Итак, после выполнения всех перечисленных операций и соединения элемента управления TAdapterGrid с адаптером TDataSetAdapter и через него с компонентом TADOTable страница начинает работать корректно и в закладке **Browser** редактора компонента TAdapterPageProducer появляются данные из таблицы Country базы данных.

Аналогичным образом можно работать и с другими элементами управления.

Редактирование данных

Для редактирования данных из таблицы Country добавим в иерархический список элементов управления компонента TAdapterPageProducer еще одну форму и в ней создадим элемент управления TAdapterFieldGroup. Он обеспечит отображение всех полей текущей записи набора данных в отдельных однострочных редакторах. При помощи свойства Adapter объект набора полей связывается с комопнентом-адаптером и через него — с компонентом набора данных.

Для навигации по набору данных необходимо добавить элемент управления TAdapterCommandGroup, который представляет собой набор кнопок, позволяющих перемещаться по записям, редактировать данные и сохранять изменения. И он должен быть связан с набором полей TAdapterFieldGroup. Для этого свойство DisplayComponent класса TAdapterCommandGroup должно ссылаться на объект набора полей.

В результате после компиляции приложения на странице DataViewer можно просматривать и редактировать данные из таблицы Country.

Обработка ошибок

В приложениях WebSnap существует возможность обрабатывать исключительные ситуации на нескольких уровнях.

На уровне всего приложения компонент TWebAppComponents предоставляет метод-обработчик

```
type
  TWebExceptionEvent = procedure (Sender: TObject; E: Exception;
  var Handled: Boolean) of object;
property OnException: TWebExceptionEvent
```

Параметр `Handled` управляет обработкой ошибки. Если он устанавливается в `True`, это означает, что исключительная ситуация будет обработана в этом методе-обработчике.

Исключительные ситуации на уровне отдельных страниц можно обрабатывать при помощи свойства

```
property Errors: TBaseAdapterErrorsList;
```

компонентов-адаптеров.

Через него разработчик в первую очередь может получить доступ к списку ошибок, возникших при работе компонента:

```
var ErrorsList: TStringList;
...
if Adapter1.Errors.ErrorCount > 0 then
  for i := 0 to Adapter1.Errors.ErrorCount - 1 do
   ErrorsList.Add(Adapter1.Errors.Errors[i].ErrorText);
```

Класс `TBaseAdapterErrorsList` имеет следующие свойства и методы.

Главное свойство

```
property Errors[I: Integer]: IAdapterError;
```

представляет собой индексированный список возникших ошибок. Параметры ошибки представлены интерфейсом `IAdapterError`.

Общее число ошибок в списке возвращает свойство

```
property ErrorCount: Integer
```

Список можно очистить методом

```
procedure ClearErrors;
```

Например:

```
if Adapter1.Errors.ErrorCount > 0
 then Adapter1.Errors.ClearErrors;
```

При создании бизнес-логики страниц приложения для отслеживания неверных действий пользователя наверняка пригодятся методы, добавляющие к списку новую ошибку:

```
procedure AddError(AException: Exception; AObject: TObject);
procedure AddError(AException: Exception; const AObjectName: string =
'');
procedure AddError(const AMessage: string; AObject: TObject; AID: Integer
= 0);
procedure AddError(const AMessage: string; const AObjectName: string =
''; AID: Integer = 0);
```

Например:

```
try
...
except
 on E: Exception do Adapter1.Errors.AddError(E);
end;
```

Аутенификация пользователей

В Web-приложениях часто бывает необходимо обеспечить аутенификацию пользователей при подключении или обращении к определенным страницам. В технологии WebSnap для этого предусмотрены специализированные компоненты, общее описание которых представлено ранее.

В качестве примера добавим к созданному тестовому приложению возможность аутентификации пользователей. Для этого необходимо.

1. Интегрировать в приложение список имен пользователей и паролей или добавить функцию, позволяющую получать такой список из внешних источников (например, базы данных или зашифрованного файла).
2. Создать страницу, содержащую элементы управления для ввода имени и пароля пользователя.
3. Обеспечить взаимодействие компонентов и страниц приложения.
4. Выделить страницы приложения, доступ к которым осуществляется только авторизованными пользователями, и настроить их.

Рассмотрим эти этапы более подробно.

Списки пользователей и сессии

Процесс аутенификации пользователя подразумевает, что приложение сможет сравнить введенные имя пользователя и пароль с имеющимися в списке пользователей. Такой список инкапсулирует компонент `TWebUserList`. В приложении SimpleWebSnap Application этот компонент необходимо перенести в модуль приложения. При щелчке на кнопке свойства `UserItems` в Инспекторе объектов появляется диалог, в котором можно редактировать список пользователей. Для каждого пользователя можно задать имя, пароль и права.

Затем, для управления сессиями добавим в модуль приложения компонент `TSessionsService`. Он не требует специальной настройки.

После регистрации информация о пользователе и текущей сессии доступна в приложении при посредстве компонента `TEndUserSessionAdapter`. Его также необходимо перенести в модуль приложения.

Обратите внимание, что после переноса в модуль страницы все три компонента автоматически были связаны с компонентом `TWebAppComponents` через соответствующие свойства. Тем самым обеспечивается единое управление всеми компонентами для аутенификации пользователей.

Создание страницы аутенификации

Страница аутенификации — это обычная Web-страница, обеспечивающая ввод имени и пароля пользователя. Поэтому для приложения Simple-WebSnap Application создадим еще одну страницу HTML, выбрав для нее компонент-продюсер `TAdapterPageProducer`, подобно странице, связанной с базой данных. Имя страницы — `UserLogin`.

Но в этом случае существует маленькое отличие — страница не должна быть доступной при создании приложения. Для этого необходимо выключить флажок **Published** в окне мастера создания страницы (см. рис. 41.3).

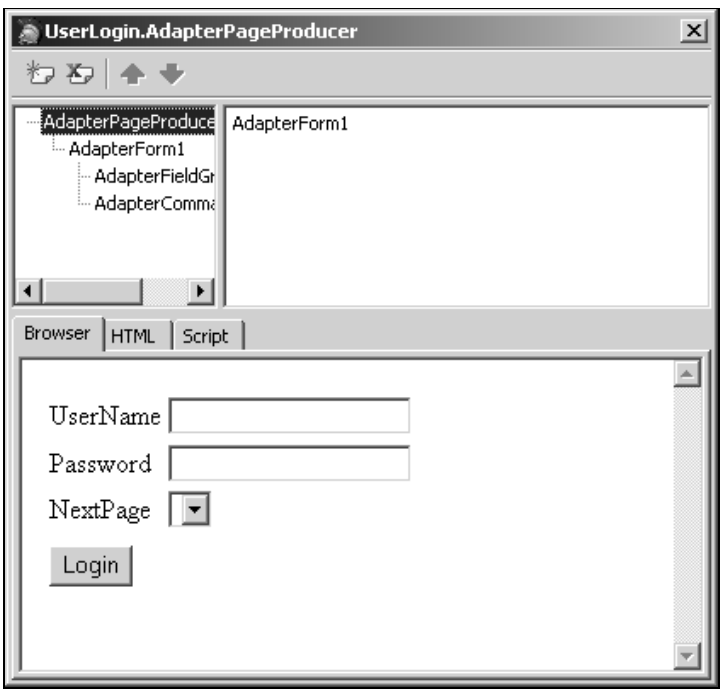


Рис. 41.13. Стандартная страница аутенификации пользователей приложения WebSnap

Затем при помощи редактора компонента TAdapterPageProducer создадим форму HTML и на ней элементы управления TAdapterFieldGroup и TAdapterCommandGroup. Свяжем между собой эти два объекта при помощи свойства DisplayComponent. Теперь в закладке **Browser** можно увидеть созданную страницу (см. рис. 41.13).

Осталось перенести в модуль страницы компонент TLoginFormAdapter, который обеспечивает правильное функционирование созданных элементов управления. И именно его необходимо указать в свойстве Adapter экземпляра класса TAdapterFieldGroup.

Теперь необходимо связать предопределенное действие Login компонента TLoginFormAdapter с нажатием кнопки **Login**. Для этого создадим для экземпляра TAdapterCommandGroup в редакторе компонента новый объект TAdapterActionButton. И в его свойстве ActionName в Инспекторе объектов выберем из списка действие Login.

И в заключение можно немного изменить шаблон страницы. Дело в том, что при создании страницы в ее шаблон автоматически добавляются два фрагмента. Один обеспечивает появление на странице ссылки Login, если пользователь еще не ввел пароль. Второй создает строку навигации по доступным страницам приложения. Если вы сохраните эти части шаблона, ничего страшного не случиться, но, на наш взгляд, они все же лишние. Поэтому шаблон страницы аутенификации в тестовом приложении выглядит как в листинге 41.3.

Листинг 41.3. Файл HTML страницы аутенификации

```
<HTML>
<HEAD>
<TITLE>
<%= Page.Title %>
</TITLE>
</HEAD>
<BODY>
<H1><%= Application.Title %></H1>
<H2><%= Page.Title %></H2>
<#STYLES><#WARNINGS><#SERVERSCRIPT>
</BODY>
</HTML>
```

После сохранения страница готова к работе.

Настройка приложения

И в завершение необходимо связать созданную страницу с процессом аутенификации в приложении и определить, какие его страницы будут доступны только через пароль.

В компоненте `TEndUserSessionAdapter` в свойстве

```
property LoginPage: string;
```

зададим имя страницы аутенификации — `UserLogin`. Теперь созданная страница будет автоматически вызываться приложением при попытке открыть защищенную страницу или при щелчке на ссылке **Login**.

Для того чтобы защитить страницу приложения, достаточно изменить параметры создания экземпляра страницы фабрикой класса. Сделаем это для страницы DataView. Во фрагменте исходного кода страницы

```
initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.AddWebModuleFactory( TWebPageModuleFactory.Create
   (TDataView, TWebPageInfo.Create([wpPublished{, wpLoginRequired}],
   '.html'), crOnDemand, caCache));
```

необходимо убрать комментарии с параметра `wpLoginRequired`.

Теперь при выполнении приложения при попытке открыть эту страницу автоматически будет вызываться страница с предложением ввести имя и пароль пользователя.

Однако проверка созданного механизма аутенификации имеет особенность. При аутенификации пользователя используется информация о сессии приложения. А она сохраняется, только если приложение работает все время между вызовами страниц.

Так как тестовое приложение имеет тип Web App Debugger, то перед обращением к главной странице MainPage из утилиты Web App Debugger необходимо вручную запустить исполняемый файл приложения.

Использование XML и XSL

Наряду с шаблонами HTML приложения WebSnap могут использовать и шаблоны XML. Для этого служит компонент-продюсер `TXSLPageProducer`. Он обеспечивает динамическую генерацию страницы на основе шаблона на языке XML с использованием шаблонов стилей XSL.

Рассказ о возможностях языков XML и XSL выходит за рамки нашего обсуждения, поэтому в приведенных далее примерах мы ограничимся простейшими шаблонами, сосредоточив внимание на возможностях техноло-

гии WebSnap по применению XML и XSL. Подробнее об XML и XSL *см. главу 13*.

Для демонстрации возможностей XML создадим новое приложение Simple XML WebSnap Application, подобно приведенному ранее в этой главе. Пусть главная страница приложения останется стандартной главной страницей HTML со всеми компонентами, создаваемыми по умолчанию, — они будут выполнять свою работу независимо от формата шаблонов других страниц.

Создадим для приложения две дополнительные страницы. И т. к. приложение должно работать с шаблонами XML, то при создании страницы в мастере (см. рис. 41.3) необходимо выбрать компонент-продюсер `TXSLPageProducer`.

Далее мы расскажем, как обеспечить генерацию страниц на основе файлов XML и XSL и на основе данных, получаемых из базы данных.

Использование файла XML и шаблона XSL

Сначала посмотрим, как можно применить файлы XML для формирования страниц приложений WebSnap.

Назовем первую страницу ListViewer. Кроме компонента-продюсера `TXSLPageProducer` она почти ничем не отличается от обычной страницы на основе шаблона HTML. Лишь в нижней части окна Редактора Delphi появилась одна дополнительная закладка для файла XSL.

Настройка страницы для шаблона XML выполняется в два этапа.

На первом этапе необходимо связать страницу с файлом XML.

Теперь перенесем в модуль страницы компонент `TXMLDocument` и зададим в его свойстве

```
property FileName: DOMString;
```

имя файла SimpleXMLDataList.xml. Этот файл содержит перечень столиц и число жителей из таблицы Country демонстрационной базы данных Delphi. Обратите внимание, что для обеспечения переносимости приложения при работе с этим свойством необходимо:

- размещать файлы XML в папке приложения и не указывать полный путь к ним;
- указывать пути к файлу через URL.

Примечание

Файл SimpleXMLDataList.xml создан на основе стандарта Microsoft XML Integration in ADO 2.5, используемом при экспорте и импорте данных для источников данных ADO.

Настроенный таким образом компонент документа XML необходимо подключить к странице. Для этого используем компонент `TXSLPageProducer` и его свойство

```
property XMLData: TComponent;
```

На втором этапе странице необходимо придать таблицу стилей XSL.

Для этого необходимо перенести в папку приложения нужный файл XSL и переименовать его в соответствии с именем файлов создаваемой страницы. В нашем случае это должен быть файл ListViewer.xsl. Еще один способ — скопировать текст стиля XSL в исходном файле и перенести его прямо в окно Редактора Delphi и затем сохранить файл XSL-страницы.

Теперь, если шаблон XML и стиль XSL имеют корректный синтаксис, в окне Редактора **HTML Result** доступен сгенерированный компонентом `TXSLPageProducer` код HTML для создаваемой страницы. А в окне **Preview** можно увидеть, как страница будет выглядеть во время выполнения приложения.

В нашем примере файл XSL (листинг 41.4) обеспечивает простейший способ отображения данных в две текстовые колонки.

Листинг 41.4. Файл XSL для страницы ListViewer приложения Simple XML WebSnap Application

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
 <xsl:template match = "*">
 <xsl:apply-templates />
 </xsl:template>
 <xsl:template match="/">
 <H2>Simple XML WebSnap Application</H2>
 <xsl:apply-templates select="//rs:data"/>
 </xsl:template>
 <xsl:template match="//rs:data">
        <TABLE BORDER="1">
             <TR>
               <TD BGCOLOR="gray" STYLE="font-weight:bold">Country</TD>
               <TD BGCOLOR="gray" STYLE="font-weight:bold">Capital</TD>
             </TR>
           <xsl:for-each select="z:row">
             <TR>
               <TD><xsl:value-of select="@Name"/></TD>
               <TD><xsl:value-of select="@Capital"/></TD>
```

```
            </TR>
          </xsl:for-each>
        </TABLE>
 </xsl:template>
</xsl:stylesheet>
```

Обратите внимание, что элементы

```
<xsl:apply-templates select="//rs:data"/>
```

и

```
<xsl:for-each select="z:row">
```

обеспечивают доступ к узлам документа XML, содержащим данные, а элементы

```
<xsl:value-of select="@Name"/>
```

и

```
<xsl:value-of select="@Capital"/>
```

предназначены для получения значения полей.

Использование набора данных и шаблона XSL

Кроме файлов XML страницы на основе XML и XSL можно создавать и для данных, хранящихся в базах данных.

Для демонстрации этой возможности технологии WebSnap создадим еще одну страницу для приложения Simple XML WebSnap Application. Назовем ее DataViewer. Также как и для предыдущего случая, в качестве компонента-продюсера выберем `TXSLPageProducer`.

Однако источником данных теперь будет набор данных ADO. Для организации доступа к данным дополнительно создадим Web-модуль данных и разместим на нем компоненты ADO `TADOConnection` и `TADOTable`. Настроим их для доступа к базе данных Microsoft Access DBDEMOS.MDB из демонстрационной базы данных Delphi. (Подробнее о технологии и компонентах ADO *см. главу 23*). Кроме этого, нам понадобится компонент `TDataSetProvider`, который обеспечит передачу данных. (Подробнее о роли компонента `TDataSetProvider` *см. главу 27*). Свяжем между собой компонент таблицы ADO и компонент `TDataSetProvider` через его свойство `DataSet`.

Затем перенесем в модуль данных создаваемой страницы DataViewer компонент `TXMLBroker` со страницы **Internet Express**. Для его свойства `ProviderName` зададим имя компонента `TDataSetProvider` из модуля данных. А компонент `TXMLBroker` должен быть связан с компонентом-

провайдером страницы. Для этого свойству `XMLData` компонента `TXSLPageProducer` присвоим ссылку на компонент `TXMLBroker`.

Итак, вся цепочка компонентов, обеспечивающих передачу данных из базы данных в страницу XML, настроена. Осталось создать стиль XSL. Это делается так же, как и для предыдущего случая. Сам файл XSL этой страницы для представления данных описывает таблицу (листинг 41.5).

Листинг 41.5. Файл XSL для страницы DataViewer приложения Simple XML WebSnap Application

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
 <xsl:template match = "*">
 <xsl:apply-templates />
 </xsl:template>
 <xsl:template match="/">
 <H2>Simple XML WebSnap Application</H2>
 <xsl:apply-templates select="//ROWDATA"/>
 </xsl:template>
 <xsl:template match="//ROWDATA">
        <TABLE BORDER="1">
             <TR>
               <TD BGCOLOR="gray" STYLE="font-weight:bold">Country</TD>
               <TD BGCOLOR="gray" STYLE="font-weight:bold">Capital</TD>
             </TR>
           <xsl:for-each select="ROW">
             <TR>
               <TD><xsl:value-of select="@Name"/></TD>
               <TD><xsl:value-of select="@Capital"/></TD>
              </TR>
           </xsl:for-each>
        </TABLE>
 </xsl:template>
</xsl:stylesheet>
```

Обратите внимание, что в этом случае компонент `TXMLBroker` передает компоненту-продюсеру `TXSLPageProducer` данные XML, которые описывают набор данных в формате пакета данных Delphi. Поэтому шаблон стиля содержит несколько иные элементы для узлов, инкапсулирующих данные:

```
<xsl:apply-templates select="//ROWDATA"/>
<xsl:for-each select="ROW">
```

Дополнительный компонент *TAdapterXMLBuilder*

Кроме стандартных компонентов WebSnap, о которых уже много рассказывалось в этой главе, для создания страниц XML в распоряжении разработчиков есть дополнительный компонент, который включен в примеры WebSnap. Исходные файлы класса `TAdapterXMLBuilder` находятся в папке ..\Demos\WebSnap\XMLBuilder. Регистрацию компонента необходимо выполнить при помощи файла XMLBuilder.reg. После установки в Палитру инструментов компонент доступен для выбора провайдеров в мастере создания Web-страниц.

Этот компонент позволяет конструировать интерфейс Web-страницы подобно компоненту `TAdapterPageProducer`. Окно редактора компонента (открывается двойным щелчком на компоненте) состоит из панели иерархического дерева элементов управления слева и панели составных частей элемента управления, выбранного в дереве.

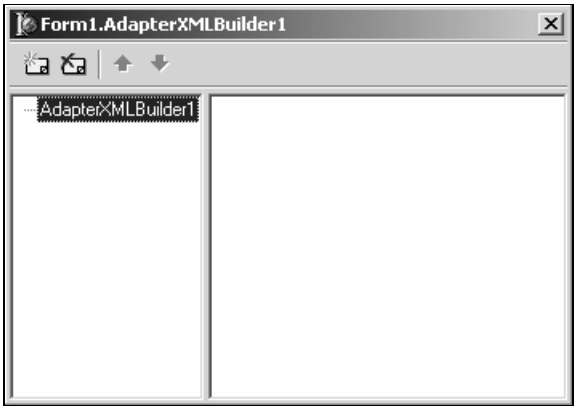


Рис. 41.14. Окно редактора компонента `TAdapterXMLBuilder`

Элементы управления можно располагать только в форме, инкапсулированной классом `TXMLForm`. Здесь доступны те же элементы управления, что и для компонента `TAdapterPageProducer`, за исключением класса-аналога `TLayoutGroup`.

Так как основой для генерации страницы в этом случае является шаблон XML, то компоненту для работы требуется и файл стиля XSL.

Свойство

```
property XSLTemplate: TXSLTemplate;
```

позволяет связать с компонентом нужный файл XSL. Однако, как вы видите, это делается не напрямую, а при посредстве еще одного компонента

`TXSLTemplate`. Он устанавливается в Палитру компонентов совместно с компонентом `TAdapterXMLBuilder`. Для указания файла XSL компонент имеет свойство

```
property FileName: DOMString;
```

В результате на основе созданного дерева элементов управления, описанных на языке XML, и шаблона стиля XSL компонент генерирует Web-страницу. Автоматически создаваемый компонентом код XML можно просмотреть в окне с закладкой **XML Tree** в Редакторе Delphi.

Резюме

Технология WebSnap обеспечивает создание широкого спектра Web-приложений:

- ISAPI/NSAPI DLL
- CGI
- Win-CGI
- Apache Shared Module (DLL)
- Web App Debugger

Обработка запросов, поступающих приложению WebSnap от Web-сервера, обеспечивается триадой компонентов.

- Компоненты-диспетчеры — управляют запросами, направляя их нужным страницам.
- Компоненты-продюсеры — обеспечивают генерацию страниц на основе имеющегося шаблона.
- Компоненты-адаптеры — позволяют связывать пользовательский интерфейс страниц с запрограммированными действиями.

При необходимости реализовать сложные функции разработчик может создавать скрипты на языках VBScript или JScript.

При разработке страниц приложений можно использовать как готовые шаблоны, созданные на языках HTML или XML, так и специализированные компоненты, позволяющие конструировать интерфейс страницы "на лету".

Приложение

Описание компакт-диска

Папка	Описание	Глава
\08_1	Пример применения компонентов `TTreeView` и `TTreeList` для просмотра информации из системного реестра Windows	8
\08_2	Простой пример разработки собственного компонента	8
\08_3	Ресурсы манифеста Windows XP	8
\09_1	Простой пример работы с компонентами меню	9
\10_1	Пример использования в приложении списка объектов на основе класса `TList`	10
\10_2	Пример использования в приложении списка строк на основе класса `TStringList`	10
\12_1	Пример приложения с компонентами Delphi для отображения графики	12
\12_2	Приложение для просмотра растровых приложений JPG, JPEG, BMP	12
\13_1	Пример приложения для просмотра документов XML на основе модели DOM	13
\13_2	Редактор документов XML. Пример использования интерфейсов `IXMLNode` и `IXMLDocument`	13
\13_3	Пример применения технологии доступа к данным ADO при работе с документами XML	13
\15_1	Пример простейшего приложения баз данных	15
\16_1	Пример использования параметров компонентов запросов SQL и взаимодействия таких компонентов на основе передачи значений параметров	16
\18_1	Приложение баз данных, реализующее отношение "один ко многим" между таблицами базы данных	15
\18_2	Приложение баз данных, демонстрирующее варианты поиска записей в таблице базы данных	15
\18_3	Пример закладок (класс `TBookmark`) в наборах данных	18

(окончание)

Папка	Описание	Глава
\19_1	Приложение баз данных, использующее компоненты синхронного просмотра	19
\20_1	Приложение баз данных, напрямую использующее API BDE для полного удаления записей из таблиц базы данных	20
\20_2	Приложение баз данных, напрямую использующее API BDE для представления данных	20
\21_1	Приложение баз данных, демонстрирующее возможности технологии dbExpress	21
\23_1	Приложение баз данных, демонстрирующее возможности технологии ADO	23
\24_1	Пример сервера и клиента COM	24
\25_1	Пример сервера и контроллера Автоматизации	25
\28_1	Пример простого распределенного приложения баз данных	28
\30_1	Пример приложения для просмотра и редактирования данных XML в виде набора данных и документа XML	30
\33_1	Пример использования компонента проекта отчета Rave Reports и разработки простых отчетов в визуальной среде Rave Reports	33
\34_1	Пример реализации настраиваемого соединения на основе компонента `TRvCustomConnection`	34
\34_2	Примеры создания различных видов отчетов в приложении Rave Reports	34
\36_1	Менеджер сертификатов. Пример применения CryptoAPI	36
\41_1	Простой пример приложения на основе технологии Web Snap	41
\41_2	Пример приложения на основе технологии Web Snap, создающего страницы XML и XSL на основе данных из баз данных	41

Предметный указатель

A

Abort 97
Action 208
Action client 208
Action Target 208
actions 961
ActiveX 734
ADO 613
AnsiString 17
API BDE 513
as 54
assembly 51

B

BDE 500
Boolean 16
brcc32.exe 154
ByteBool 16

C

Cardinal 15
CGI (Common Gateway Interface) 959
CLSID (Class Identifier) 673
CoClass 676
CoCreateInstance 674
CoGetClassObject 675
COM (Component Object Model) 665
Comctl32.dll 142
ConnectionString 622
cookies 991

D

DataSnap 792, 795
dbExpress 547
default 33
Device — dependent bitmap 293
Device — Independent Bitmap 294
DIB 294
dispinterface 713
Double 16
DTD 325
dynamic 39, 44
Dynamic Method Table 44

E

EAbort 96
EAssertionFailed 97
EFCreateError 273
EFOpenError 273
EInOutError 82
EIntError 85
EReadError 273
EReconcileError 813
EWriteError 273
ExceptAddr 91
Exception 78
ExceptObject 91
Extended 16

F

family 51

G

GUID (Globally Unique IDentifier) 671, 680

I

IAccessor 618
IAppServer 772, 775, 793
IClassFactory 675, 681, 682
IClassFactory2 682
IColumnsInfo 618
ICommand 619
ICommandPrepare 620
ICommandProperties 620
ICommandText 620
ICommandWithParameters 620
IConnectionPointContainer 717
IConvertType 619
IDBCreateSession 618
IDBInitialize 617
IDBProperties 617
IDispatch 708
IDL (Interface Definition Language) 669
IDL CORBA 689
IDL Microsoft 689
IDOMDocument 332
IDOMElement 332
IDOMNode 332
IID (Interface Identifier) 671
IInterface 679
inherited 30, 43
Integer 15
Invoke 708, 720
IOpenRowset 618
IOResult 82
IProvideClassInfo 679
IProviderSupport 786, 794
IRowset 618
IRowsetChange 619
IRowsetIdentity 619
IRowsetIndex 619
IRowsetInfo 618
IRowsetLocate 619
IRowsetUpdate 619
is 54
ISAPI (Internet Services API) 959
ISourcesRowset 616
ISQLCommand 577
ISQLConnection 576
ISQLCursor 579
ISQLDriver 575
ISupportErrorInfo 724
ITransaction 618
ITransactionJoin 618
ITransactionLocal 618
ITransactionObject 618
ITypeInfo 676
ITypeLib 676
IUnknown 672, 677, 682

J

JPEG 298

L

LongBool 16

M

merge 204
Microsoft ActiveX Data Objects 613

N

NSAPI (Netscape Services API) 960

O

OLE DB 613
OnException 89
overload 46
override 44

P

parser 325
private 47

property 31
protected 48
public 47
published 48

R

raise 81, 86, 92
Real 16
Real48 16
reintroduce 47
RPC (Remote Procedure Call) 673

S

sealed 40
Self 28, 36
Sender 35
ShortString 17
ShowException 88
Single 16
SQL Links 500

T

TAction 208, 211
TActionList 208
TADOCommand 652
TADOConnection 622
TADODataSet 650
TADOQuery 651
TADOStoredProc 652
TADOTable 650
TAnimate 318
TApplication 109
TAutoIntfObject 723
TAutoObject 715
TAutoObjectFactory 722
TBDEDataSet 529
TBitmap 293
TBrush 282
TCanvas 76, 282
TCaption 70
TChart 493
TClass 54
TClientDataSet 793, 794, 826
TClipboard 310
TCollection 246
TCollectionItem 247
TColor 70
TColumn 468
TComClassManager 684
TCommonCalendar 171
TComObject 676, 677, 724
TComObjectFactory 681, 682
TComponent 63, 128
TComServer 685
TConnectionBroker 769
TControl 67, 128
TControlState 71
TControlStyle 72
TCursor 70
TCustomADODataSet 636
TCustomControl 75
TCustomForm 128
TCustomSQLDataSet 560
TCustomTabControl 145
TCustomWebDispatcher 979
TDatabase 524
TDataSet 398, 400, 424, 795
TDataSetField 809
TDatasetPageProducer 993
TDataSetProvider 771, 781, 794
TDatasetTableProducer 995
TDataSource 387, 389, 393, 419, 492
TDateTimePicker 171
TDBChart 493
TDBCheckBox 486
TDBComboBox 487
TDBCtrlGrid 479
TDBDataSet 534
TDBGrid 468
TDBGridColumns 468
TDBImage 488
TDBListBox 487
TDBLookupComboBox 492
TDBLookupControl 490
TDBLookupListBox 492
TDBMemo 487

TDBNavigator 396, 482
TDBRadioGroup 486
TDBRichEdit 488
TDBText 485
TDCOMConnection 759
TField 432
TFileStream 269
TFont 70, 280
TFontDialog 281
TForm 128
TGraphic 288
TGraphicControl 76
TGUID 680
THandleStream 269
TIBDatabase 587
TIBDatabaseInfo 609
TIBDataSet 601
TIBEvents 607
TIBGeneratorField 601
TIBQuery 599
TIBSQL 604
TIBSQLMonitor 611
TIBStoredProc 601
TIBTable 599
TIBTransaction 590
TIcon 293
TImage 301
TImageList 153, 200
TIndexDef 413
TIndexDefs 413
TInterfacedObject 680
TISAPIApplication 959
TJPEGImage 298
TList 238
TListView 161
TLocalConnection 767
TMainMenu 204
TMemoryStream 271
TMenu 200
TMenuItem 200
TMetafile 292
TMonthCalendar 169
TNotifyEvent 35
TObject 39, 54, 60
TOpenPictureDialog 302
TPageControl 144
TParam 418
TParameter 648
TParameters 647
TPen 281
TPicture 290
TPopupMenu 207
TQuery 542
TQueryTableProducer 995
TRect 69
TRemoteDataModule 775
try..except 83
try..finally 83
TSavePictureDialog 302
TScreen 311
TScrollingWinControl 128
TSearchRec 265
TSharedConnection 768
TSimpleDataSet 567
TSimpleObjectBroker 765
TSocketConnection 761
TSQLConnection 550, 556, 558
TSQLDataSet 563
TSQLMonitor 579
TSQLQuery 564
TSQLQueryTableProducer 995
TSQLStoredProc 565
TSQLTable 563
TStatusBar 173
TStatusPanel 173
TStoredProc 545
TStream 267
TStringList 230
TStrings 229
TStringStream 272
TTabControl 144
TTable 392
TTabSheet 144
TToolBar 149
TToolButton 149
TTreeView 155
TTypedComObject 679
TTypedComObjectFactory 682, 684
TWebActionItem 964
TWebApplication 962, 979
TWebConnection 764

TWebDispatcher 963, 976
TWebModule 979
TWebRequest 965
TWebResponse 965
TWinControl 73, 128
TXMLBroker 828
TXMLTransform 820
TXMLTransformClient 826
TXMLTransformProvider 824

U

Unassigned 25

V

VarArrayCreate 26
VarArrayOf 26
virtual 39, 44
Virtual Method Table 44

W

Web модуль 979
Web-запрос 965
Web-ответ 965
Web-сервер 956
WideString 18
WinCGI 960
WordBool 16
wrapper 745

X

XML 323
XML-анализатор 325
XSL 325
XSLT 325
XSQLDA 596
XSQLVAR 597

А

Автоматизация 707
Агрегатные поля 807
Агрегаты 804
Агрегаты-объекты 805
Асимметричное шифрование 924
Атрибуты файла 265
Аутентификация 925

Б

Библиотека:
  COM 668, 673
  типов 669, 676, 687, 697
Битовая карта 293
Блок защиты ресурсов 87
Буфер Delta 798

В

Вариант 25
Виртуальные методы 41
Вложенные наборы данных 809
Всплывающее меню 207
Вызов удаленных процедур 673
Вычисляемое поле 439

Г

Главное меню 204
Глобальный объект приложения 977
Глобальный уникальный идентификатор 671
Группа проектов 105
Группировка агрегатов 809

Д

Действие 208, 964, 961, 976
Дейтаграмма 914
Делегирование 36
Дескриптор окна 73, 117, 124
Деструктор 29
Диапазон 464
Динамические поля 430

Диспетчер проектов 106
Диспетчерский идентификатор 709
Диспинтерфейс 708
Дочерний класс 39
Драйверы dbExpress 550

З

Закладка 461, 537
Запись 23
Зашифрованная последовательность данных (ciphertext) 922
Значок 293

И

Идентификатор:
  интерфейса 671
  класса 673
Индекс 412
Инкапсуляция 38
Интерфейс 666, 670
  IDispatch 712
  диспетчеризации 708, 713
  дуальный 710, 714
Исходный текст (plaintext) 922

К

Кисть 282
Класс 28
Клиент многозвенных приложений 792
Коллекции 245
Колонка 468
Команда ADO 619
Компонент-продюсер 976, 982
Компоненты доступа к данным 389
Компоненты отображения данных 395, 466
Конструктор 29
Контроллер Автоматизации 711, 725
Конфигурация BDE 504

М

Маршалинг 668
  диспинтефрейсов 710
Маска 449
Массив:
  динамический 19, 21
  многомерный 22
  статический 19
Меню 200
Метафайл 292
Метод 28, 32
  абстрактный 42
  виртуальный 44
  динамический 44
  класса 29, 55
  перегружаемый 45
  статический 43
Метод-обработчик событий 35
Множество 19
Модель взаимодействия открытых систем (OSI) 908
Модуль 102
Моникер 668, 675

Н

Набор данных 387, 398
Наследование 39
Номер порта 918

О

Область видимости 47
Обработка исключительных ситуаций 77
Обработка сообщений 118
Объект 29
  COM 666, 669, 676, 689
  Автоматизации 711, 714
  поля 427
Объект-источник данных ADO 617
Объект-команда ADO 619

Объект-набор рядов ADO 618
Объектный тип 28
Объект-перечислитель ADO 616
Объект-приемник 717
Объект-сессия ADO 617
Объект-транзакция ADO 618
Объекты в списках 232
Ограничения данных 447
Опережающее объявление класса 28
Отложенный ввод-вывод 262
Отношение "многие-ко-многим" 455
Отношение "один-ко-многим" 453
Отображаемым файлом 275
Очередь сообщений 117

П

Пакет данных 375, 796
Панель инструментов 149
Папка обмена 310
Параметры запроса SQL 417
Первичный индекс 415, 801
Перо 281
Подписанный хэш 927
Позднее связывание 44, 708
Поиск данных 456
Поиск файла 266
Поле 28, 31
Полиморфизм 41
Пользовательский интерфейс 387
Поставщик криптографических услуг 934
Потоки 267
Провайдер ADO 615, 620
Провайдер данных 781
Проект 101
Пролог 328
Процессор баз данных 500
Псевдоним базы данных 504

Р

Раннее связывание 708
Редактор библиотеки типов 687
Репозиторий 108
Родительский класс 39

С

Свойство 31
Сервер COM 666, 672
Сервер:
  Автоматизации 711, 724
  приложения 771
Серверное Web-приложение 956
Сертификат 929
Симметричное шифрование 924
Синхронный просмотр данных 489
Слияние меню 204
Событие 34
Соответствие типов 41
Списки 229
  указателей 237
Стандарт DN 937
Статические поля 430
Стиль формы 139
Страничный файл 275

Т

Технология Web Broker 974
Тип:
  вещественный 15
  перечисляемый 17
  поддиапазон 17
  строковый 17
Типы:
  данных 442
  полей 433, 436
  простые 14
  структурные 18
Тонкий клиент 756
Трансформационный файл 381
  XML 374
Трехзвенная модель 755
Тэг 324

У

Удаленный модуль данных 757, 775
Указатель на метод 36
Устройства контекста 279
Утилита XML Mapper 376, 377

Ф

Фабрика класса 669, 675, 681
  автоматизации 722
Файл DPR 30
  с расширением BDSGROUP 105
  с расширением DCU 105
  с расширением DFM 103
  с расширением DPR 101
  с расширением RES 105
  с расширнием PAS 102
Файловые переменные 250
Файлы, отображаемые в память 315
Фильтр 459, 529
Форма 103, 124
Функции:
  блочного ввода/вывода 254
  ввода/вывода 251
Функция IOResult 265

Х

Хэш 926

Ц

Цвет 70
Целостность данных 926

Ш, Э

Шрифт 70, 280
Экземпляр класса 29