bhv®

Петр Дарахвелидзе
Евгений Марков

Разработка Web-служб средствами Delphi

+Дискета

- Архитектура распределенных приложений баз данных
- Использование XML и протокола SOAP
- Распределенные приложения и Web-службы
- Криптография и безопасность данных

МАСТЕР_ПРОГРАММ_

Петр Дарахвелидзе
Евгений Марков

Разработка Web-служб средствами Delphi

Санкт-Петербург
«БХВ-Петербург»
2003

УДК 681.3.06
ББК 32.973.26-018
Д20

Дарахвелидзе П. Г., Марков Е. П.

Д20 Разработка Web-служб средствами Delphi. — СПб.: БХВ-Петербург, 2003. — 672 с.: ил.

ISBN 5-94157-142-9

В книге рассматриваются принципы функционирования Web-служб и инструменты для их создания и отладки, а также способы обеспечения связи и устойчивой работы распределенных приложений с помощью протокола SOAP. Подробно обсуждаются базовые технологии взаимодействия объектов, созданных в различных средах, способы разработки приложений баз данных, современные методы создания Internet-приложений, в том числе с применением языка XML. Большое количество примеров, приведенных в книге, и прилагаемая дискета помогут легче и быстрее освоить материал книги.

Для программистов

УДК 681.3.06
ББК 32.973.26-018

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Анна Кузьмина*
Редактор	*Татьяна Коротяева*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.10.02.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 54,18.
Тираж 4000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-142-9

© Дарахвелидзе П. Г., Марков Е. П., 2002
© Оформление, издательство "БХВ-Петербург", 2002

Содержание

Введение **1**

COM и COM+ 2
DataSnap 2
XML 3
SOAP 3
WebSnap и WebBroker 4
Криптография и безопасность данных 4

Часть I. Приложения COM и COM+ **5**

Глава 1. Механизмы COM в Delphi **7**

1.1. Базовые понятия 8
1.1.1 Объект 11
1.1.2. Интерфейс 12
1.1.3. Интерфейс *IUnknown* 14
1.1.4. Сервер 14
1.1.5. Библиотека COM 15
1.1.6. Фабрика класса 17
1.1.7. Библиотека типов 17
1.2. Объекты COM в Delphi 18
1.2.1. Класс *TComObject* 19
1.2.2. Класс *TTypedComObject* 21
1.2.3. Интерфейс *IUnknown* в Delphi 21
1.2.4. Тип глобального идентификатора 22
1.2.5. Класс *TInterfacedObject* 22
1.3. Фабрика класса в Delphi 23
1.3.1. Класс *TComObjectFactory* 23
1.3.2. Класс *TTypedComObjectFactory* 26
1.3.3. Класс *TComClassManager* 26

1.4. Сервер COM в Delphi 26
1.4.1. Класс *TComServer* 27
1.5. Библиотека типов в Delphi 28
1.6. Простой объект COM в составе внутреннего сервера 31
1.6.1. Создание объекта 31
1.7. Создание методов интерфейса 38
1.7.1. Регистрация внутреннего сервера 45
1.7.2. Использование интерфейсов внутреннего сервера COM 45
Резюме 48

Глава 2. Технология Автоматизация 49

2.1. Базовые понятия технологии Автоматизация 50
2.1.1. Интерфейсы Автоматизации 50
Интерфейс *IDispatch* и диспинтерфейсы 50
Дуальные интерфейсы 52
2.1.2. Библиотека типов 52
2.1.3. Маршалинг интерфейсов Автоматизации 52
2.1.4. Объект Автоматизации 53
2.1.5. Сервер Автоматизации 53
2.1.6. Контроллер Автоматизации 53
2.2. Реализация Автоматизации в Delphi 53
2.2.1. Интерфейсы автоматизации 54
Интерфейс *IDispatch* 54
2.2.2. Интерфейсы диспетчеризации 55
2.2.3. Дуальные интерфейсы 56
2.2.4. Объект Автоматизации 56
Класс *TAutoObject* 57
Обработка событий объекта Автоматизации 59
Создание методов — аналогов событий 62
Создание класса-оболочки 63
Инициализация объекта Автоматизации 63
2.2.5. Фабрика класса 65
Класс *TAutoIntfObject* 66
2.2.6. Сервер Автоматизации 66
2.2.7. Контроллер Автоматизации 67
2.3. Пример приложения Автоматизации 69
2.3.1. Сервер Автоматизации 69
2.3.2. Контроллер Автоматизации 72
Резюме 74

Глава 3. Компоненты ActiveX 75

3.1. Как работают элементы управления ActiveX 76
3.1.1. Контейнеры и регистрация элементов управления ActiveX 79

3.1.2. Предоставление методов....80
3.1.3. События....80
3.1.4. Свойства....81
3.1.5. Страницы свойств....81
3.1.6. Лицензирование....81
3.2. Реализация компонентов ActiveX в Delphi....82
3.2.1. Класс компонента ActiveX....83
3.2.2. Фабрика класса компонента ActiveX....84
3.2.3. Среда разработки Delphi как контейнер ActiveX....85
3.2.4. Регистрация компонентов ActiveX....85
3.3. Использование готовых компонентов ActiveX....85
3.3.1. Инсталляция готовых элементов управления AciveX....86
3.3.2. Деинсталляция готовых элементов управления AciveX....87
3.3.3. Пример инсталляции элемента управления *TWebBrowser*....87
3.4. Разработка собственных компонентов ActiveX....91
3.4.1. Превращение компонентов Delphi в компоненты ActiveX....92
3.4.2. Преобразование форм в формы ActiveX....93
Резюме....99

Глава 4. Технология COM+ (Microsoft Transaction Server)....100

4.1. Как работает MTS....101
4.1.1. Объект MTS....102
4.1.2. Транзакции....103
4.1.3. Контекст объекта MTS....104
4.1.4. Безопасность данных в MTS....105
Декларативная защита данных....105
Программная защита данных....106
4.1.5. Ресурсы....106
Активизация Just-in-time....106
Пулинг ресурсов....107
Освобождение ресурсов....107
Пулинг объектов....108
4.2. Создание приложений MTS в Delphi....109
4.2.1. Объекты транзакций MTS....109
Класс *TMTSAutoObject*....109
Создание объекта транзакции....111
Типы потоковой модели....113
Поведение объектов MTS в транзакциях....113
4.2.2. Удаленный модуль данных MTS....114
4.2.3. Распределители ресурсов....115
4.3. Тестирование и установка MTS компонентов....118
4.4. Оптимизация работы с MTS....120
4.4.1. Блокировка транзакций....120

4.4.2. Действия MTS ... 121
4.5. Пример простого объекта транзакции ... 121
4.5.1. Создание сервера ... 121
4.5.2. Создание клиента ... 123
4.5.3. Пример создания клиента и сервера в случае распределенной транзакции ... 124
Создание серверов ... 124
Разработка объекта распределенной транзакции ... 125
Создание клиента ... 127
Резюме ... 127

Часть II. Технологии доступа к данным ... 129

Глава 5. Архитектура приложений баз данных ... 131

5.1. Общая струкрура приложения баз данных ... 133
5.1.1. Как работает приложение баз данных ... 133
5.1.2. Модуль данных ... 136
5.1.3. Подключение данных ... 137
5.1.4. Настройка компонента *TDataSource* ... 138
5.1.5. Отображение данных ... 140
5.2. Набор данных ... 141
5.2.1. Абстрактный набор данных ... 141
Навигация по набору данных ... 142
Поля набора данных ... 143
Редактирование набора данных ... 144
Обработка событий набора данных ... 145
5.2.2. Стандартные компоненты ... 146
Компонент таблицы ... 146
Компонент запроса ... 149
Компонент хранимой процедуры ... 151
5.2.3. Состояния набора данных ... 152
5.3. Индексы ... 155
5.3.1. Механизм подключения индексов ... 156
5.3.2. Список описаний индексов ... 156
5.3.3. Описание индекса ... 157
5.4. Параметры запросов и хранимых процедур ... 159
5.4.1. Класс *TParams* ... 162
5.4.2. Класс *TParam* ... 163
5.5. Механизмы управления данными ... 165
5.5.1. Связанные таблицы ... 165
Отношение "один-ко-многим" ... 165
Отношение "многие-ко-многим" ... 166
5.6. Поиск данных ... 166
5.7. Фильтры ... 167

5.8. Быстрый переход к помеченным записям ... 169
5.9. Поля ... 170
5.10. Объекты полей ... 171
5.10.1. Статические и динамические поля ... 172
5.10.2. Класс *TField* ... 174
5.10.3. Виды полей ... 178
Поля синхронного просмотра ... 179
Вычисляемые поля ... 181
Внутренние вычисляемые поля ... 182
Агрегатные поля ... 183
Объектные поля ... 183
5.10.4. Типы данных ... 184
Резюме ... 186

Глава 6. Технология dbExpress ... 187

6.1. Доступ к данным dbExpress ... 188
6.2. Драйверы доступа к данным ... 190
6.3. Соединение с сервером баз данных ... 190
6.4. Управление наборами данных ... 195
6.5. Транзакции ... 197
6.6. Использование компонентов наборов данных ... 198
6.6.1. Класс *TCustomSQLDataSet* ... 199
6.6.2. Компонент *TSQLDataSet* ... 201
6.6.3. Компонент *TSQLTable* ... 202
6.6.4. Компонент *TSQLQuery* ... 203
6.6.5. Компонент *TSQLStoredProc* ... 204
6.7. Компонент *TSQLClientDataSet* ... 205
6.7.1. Соединение с сервером БД ... 206
6.7.2. Сохранение изменений ... 206
6.7.3. Работа с записями ... 208
6.7.4. Обработка исключительных ситуаций ... 208
6.8. Способы редактирования данных ... 209
6.9. Интерфейсы dbExpress ... 213
6.9.1. Интерфейс *ISQLDriver* ... 213
6.9.2. Интерфейс *ISQLConnection* ... 214
6.9.3. Интерфейс *ISQLCommand* ... 215
6.9.4. Интерфейс *ISQLCursor* ... 216
6.10. Отладка приложений с технологией dbExpress ... 217
6.11. Распространение приложений с технологией dbExpress ... 219
Резюме ... 220

Глава 7. Использование ADO средствами Delphi ... 221

7.1. Основы ADO ... 221
7.1.1. Перечислители ... 224

7.1.2. Объекты соединения с источниками данных ... 225
7.1.3. Сессия ... 225
7.1.4. Транзакции ... 226
7.1.5. Наборы рядов ... 226
7.1.6. Команды ... 227
7.2. Провайдеры ADO ... 228
7.3. Реализация ADO в Delphi ... 229
7.3.1. Компоненты ADO ... 229
7.3.2. Механизм соединения с хранилищем данных ADO ... 230
7.4. Компонент *TADOConnection* ... 230
7.4.1. Настройка соединения ... 231
7.4.2. Управление соединением ... 236
7.4.3. Доступ к связанным наборам данных и командам ADO ... 239
7.4.4. Объект ошибок ADO ... 241
7.4.5. Транзакции ... 241
7.5. Наборы данных ADO ... 243
7.5.1. Класс *TCustomADODataSet* ... 244
Набор данных ... 244
Курсор набора данных ... 245
Локальный буфер ... 247
Состояние записи ... 248
Фильтрация ... 249
Поиск ... 250
Сортировка ... 251
Команда ADO ... 251
Групповые операции ... 252
7.5.2. Параметры ... 253
Класс *TParameters* ... 254
Класс *TParameter* ... 255
7.5.3. Компонент *TADODataSet* ... 256
7.5.4. Компонент *TADOTable* ... 257
7.5.5. Компонент *TADOQuery* ... 258
7.5.6. Компонент *TADOStoredProc* ... 258
7.6. Команды ADO ... 258
7.7. Объект ошибок ADO ... 260
7.8. Пример приложения ADO ... 261
7.8.1. Соединение с источником данных ... 265
7.8.2. Групповые операции ... 266
7.8.3. Фильтрация ... 266
7.8.4. Сортировка ... 266
Резюме ... 266

Часть III. Распределенные приложения баз данных 269

Глава 8. Технология DataSnap. Механизмы удаленного доступа 271

8.1. Структура многозвенного приложения в Delphi....272
8.2. Трехзвенное приложение в Delphi....275
8.2.1. Сервер приложения....276
8.2.2. Клиентское приложение....277
8.3. Механизм удаленного доступа к данным DataSnap....278
8.3.1. Компонент *TDCOMConnection*....279
8.3.2. Компонент *TSocketConnection*....280
8.3.3. Компонент *TWebConnection*....283
8.3.4. Компонент *TCORBAConnection*....284
8.4. Вспомогательные компоненты — брокеры соединений....285
8.4.1. Компонент *TSimpleObjectBroker*....286
8.4.2. Компонент *TLocalConnection*....287
8.4.3. Компонент *TSharedConnection*....288
8.4.4. Компонент *TConnectionBroker*....288
Резюме....289

Глава 9. Сервер приложения....290

9.1. Структура сервера приложения....291
9.2. Интерфейс *IAppServer*....293
9.3. Удаленные модули данных....295
9.3.1. Удаленный модуль данных для сервера Автоматизации....296
9.3.2. Дочерние удаленные модули данных....301
9.4. Провайдеры данных....302
9.5. Интерфейс *IProviderSupport*....306
9.6. Регистрация сервера приложения....307
9.7. Пример простого сервера приложения....307
9.7.1. Главная форма сервера приложения....308
9.7.2. Главный удаленный модуль данных....309
9.7.3. Дочерний удаленный модуль данных....310
9.7.4. Регистрация сервера приложения....311
Резюме....312

Глава 10. Клиент многозвенного распределенного приложения....313

10.1. Структура клиентского приложения....314
10.2. Клиентские наборы данных....316
10.3. Компонент *TClientDataSet*....317
10.3.1. Получение данных от компонента-провайдера....318
10.3.2. Кэширование и редактирование данных....320
10.3.3. Управление запросом на сервере....322
10.3.4. Использование индексов....323
10.3.5. Сохранение набора данных в файле....325

10.3.6. Работа с данными типа BLOB ... 325
10.3.7. Представление данных в формате XML ... 326
10.4. Агрегаты ... 326
10.4.1. Объекты-агрегаты ... 327
10.4.2. Агрегатные поля ... 329
10.4.3. Группировка полей и использование индексов ... 330
10.5. Вложенные наборы данных ... 331
10.6. Дополнительные свойства полей клиентского набора данных ... 332
10.7. Обработка ошибок ... 332
10.8. Пример "тонкого" клиента ... 336
10.8.1. Соединение клиента с сервером приложения ... 338
10.8.2. Наборы данных клиентского приложения ... 339
Резюме ... 340

Часть IV. Основы разработки Internet-приложений ... 341

Глава 11. Сокеты ... 343

11.1. Введение в архитектуру сетей ... 344
11.1.1. Модель OSI ... 344
11.1.2. Физический уровень ... 346
11.1.3. Протоколы канального уровня ... 347
Управление доступом к среде ... 348
Управление логическим каналом (Logical Link Control) ... 349
11.1.4. Функции сетевого уровня ... 350
11.1.5. Транспортный уровень ... 353
11.1.6. Концепции сессионного уровня, уровней представления и приложений ... 354
Сессионный (session) уровень ... 355
Уровень представления ... 355
Уровень приложения ... 355
11.2. Модель клиент/сервер и сокеты ... 356
11.2.1. Microsoft Windows и сокеты ... 357
11.3. Работа на уровне сокетов в Delphi ... 358
11.3.1. Компоненты Delphi, инкапсулирующие сокеты ... 358
11.3.2. Установление соединения ... 359
11.3.3. Синхронизация сокетов ... 360
11.3.4. Класс *TServerWinSocket* ... 361
11.3.5. Компонент *TServerSocket* ... 364
11.3.6. Класс *TClientWinSocket* ... 364
11.3.7. Компонент *TClientSocket* ... 365
Резюме ... 366

Глава 12. Криптографическая защита информации в Internet.....367

12.1. Основные термины и понятия криптографии 368
12.1.1. Секретность.....368
12.1.2. Аутентификация 370
12.1.3. Целостность.....371
12.2. Цифровые подписи, сертификаты и их применение 372
12.2.1. Использование *CryptoAPI*.....378
Структура *CryptoAPI* 379
Пример использования *CryptoAPI* — менеджер сертификатов.....381
12.2.2. Служба сертификатов Microsoft 386
12.3. Протоколы Internet для защищенных соединений.....387
12.3.1. Настройка SSL на стороне сервера IIS 5.....390
12.3.2. Настройка протокола SSL на клиентской стороне 393
Резюме.....395

Глава 13. Потоки и процессы 397

13.1. Обзор потоков 398
13.1.1. Потоки и процессы 399
13.1.2. Фоновые процедуры, или как обойтись без потоков.....400
13.1.3. Приоритеты потоков.....400
13.2. Класс *TThread*.....404
13.3. Пример создания многопоточного приложения в Delphi.....408
13.4. Проблемы синхронизации потоков.....412
13.4.1. Тупики 413
13.4.2. Гонки.....413
13.5. Средства синхронизации потоков.....414
13.5.1. Событие 415
13.5.2. Взаимные исключения.....417
13.5.3. Семафор.....417
13.5.4. Критическая секция 418
13.5.5. Процесс. Порождение дочернего процесса.....419
13.5.6. Поток 420
13.5.7. Консольный ввод.....420
13.5.8. Оповещение об изменении в файловой системе.....421
13.6. Локальные данные потока.....422
13.7. Как избежать одновременного запуска двух копий одного приложения 423
Резюме.....424

Часть V. Данные XML в распределенных приложениях 425

Глава 14. Использование XML 427

14.1. Что такое XML и для чего он предназначен 427
14.2. Основы синтаксиса XML 430
14.2.1. Пролог 431
14.2.2. Определение 433
14.2.3. Тело документа. Корневой элемент 434
14.3. Объектная модель документа 435
14.3.1. Интерфейсы семейства *IDOMNode* 436
Свойства *nodeType*, *nodeName* и *nodeValue* 437
Свойства и методы, управляющие другими вершинами 439
Пространства имен 441
Интерфейс *IDOMDocument* 441
14.3.2. Пример создания приложения, использующего модель DOM 442
14.4. Реализация модели DOM в Delphi 445
Модуль *Xmldom* 446
Модуль *Msxml* 446
Модуль *Msxmldom* 447
Модуль *XMLIntf* 447
Модуль *XMLDoc* 447
14.4.1. Интерфейс *IXMLNode* и его отличия от стандарта DOM 447
14.4.2. Взаимосвязь между всеми интерфейсами 450
14.4.3. Загрузка XML 451
Асинхронная загрузка 453
Функции, создающие экземпляр документа 453
14.4.4. Обработка ошибок анализатора 454
Пример использования интерфейсов *IXMLNode* и *IXMLDocument* 456
14.5. Анализатор MSXML, или Microsoft XML Core Services 469
Резюме 476

Глава 15. Использование данных в формате XML 477

15.1. Преобразование данных в формате XML 477
15.1.1. Схема преобразования данных XML 478
15.1.2. Формат пакета данных Delphi 479
15.1.3. Инструментарий преобразования данных XML 480
15.2. Утилита XML Mapper 481
15.2.1. Выбор исходного файла 482
15.2.2. Создание пакета данных и документа XML и сохранение преобразованных данных 483

15.2.3. Связывание элементов XML и полей пакета данных ... 484
15.2.4. Создание трансформационного файла и преобразование данных ... 485
15.3. Преобразование данных XML в распределенных приложениях ... 486
15.4. Использование данных XML в распределенных приложениях ... 489
15.4.1. Данные XML в клиентском наборе данных ... 491
15.4.2. Данные XML в документе XML ... 491
15.4.3. Данные XML на странице HTML ... 493
15.5. Пример приложения, использующего данные XML ... 495
Резюме ... 498

Часть IV. Распределенные приложения и Web-службы ... 499

Глава 16. Серверные Web-приложения. Технология Web Broker ... 501

16.1. Публикация данных в Internet. Web-серверы ... 502
16.2. Введение в интерфейсы CGI и ISAPI ... 505
16.3. Типы серверных Web-приложений ... 507
16.4. Структура серверного Web-приложения в Delphi ... 508
16.4.1. Глобальный объект приложения ... 511
16.4.2. Web-модуль ... 513
16.4.3. Действия ... 516
16.4.4. Запросы и ответы ... 517
16.4.5. Компоненты-продюсеры ... 518
16.5. Страницы HTML в серверных Web-приложениях ... 520
16.5.1. Отображение данных ... 520
16.5.2. Ввод и редактирование данных ... 523
16.6. Cookies ... 527
16.7. Использование баз данных ... 528
16.7.1. Публикация записей таблиц баз данных ... 529
16.7.2. Генерация отчетов ... 530
16.7.3. Редактирование данных ... 535
16.8. Пример простого серверного Web-приложения ... 536
16.8.1. Главное окно приложения ... 537
16.8.2. Просмотр таблицы *Country* ... 538
16.8.3. Тестирование приложения ... 541
16.8.4. Перенос приложения на платформу ISAPI ... 542
Отладка ISAPI-приложений ... 542
Резюме ... 546

Глава 17. Протокол SOAP и Web-службы. Клиентская часть ... 547

17.1. Почему SOAP? ... 547
17.1.1. Delphi и *TSomeConnection* — потенциальные проблемы ... 548

17.2. SOAP: краткое содержательное описание ... 549
17.2.1. Web-службы ... 551
17.2.2. Описание протокола WSDL ... 552
17.3. Архитектура Web-служб в Delphi ... 557
17.4. Клиент Web-службы ... 558
17.4.1. Генерация интерфейса Web-службы ... 559
17.4.2. O, RIO, RIO ... 563
17.4.3. Как связаться с нужной службой ... 565
17.4.4. Решение коммуникационных проблем ... 566
17.4.5. Что такое *Invoke Registry*. Регистрация интерфейсов и типов данных ... 569
Резюме ... 572

Глава 18. Серверная часть Web-службы. Совместимость ... 573

18.1. Создание тестового примера — службы *SimpleEcho* ... 573
18.2. Назначение и настройки компонентов серверной части ... 576
18.2.1. Компонент *TWSDLHTMLPublish* ... 577
18.2.2. Компоненты *THTTPSoapDispatcher* и *THTTPSoapPascalInvoker* ... 580
18.2.3. Клиент службы *SimpleEcho* ... 581
18.2.4. Обработка исключительных ситуаций ... 582
18.2.5. Несколько слов об использовании *TSOAPConnection* ... 584
18.3. Средства разработки для SOAP: подход Microsoft ... 585
18.3.1. Краткий обзор архитектуры Web-служб на SOAP Toolkit ... 586
18.3.2. Сценарий клиентской части ... 588
18.3.3. Создание COM-объекта *SimpleEchoCOM* и публикация Web-службы на его основе ... 590
18.4. Использование утилиты SOAP Trace ... 593
Резюме ... 595

Глава 19. Технология WebSnap ... 596

19.1. Структура приложения WebSnap ... 596
19.1.1. Обработка запросов ... 598
19.1.2. Web-модули ... 599
Модуль данных ... 600
Модуль страницы ... 601
Модуль приложения ... 602
19.1.3. Компоненты уровня приложения ... 603
Управление компонентами приложения ... 603
Адаптер и глобальная переменная приложения ... 605
Информация о пользователях ... 606
Сессии ... 608
Управление файлами приложения ... 609

19.1.4. Компоненты-диспетчеры ... 609
19.1.5. Компоненты-продюсеры ... 611
19.1.6. Компоненты-адаптеры ... 611
19.2. Создание приложения WebSnap ... 615
19.3. Конструирование интерфейса и управление данными ... 618
19.3.1. Навигация по страницам приложения ... 620
19.3.2. Использование полей и действий компонентов-адаптеров ... 621
19.3.3. Взаимодействие с базами данных ... 624
Специализированные элементы управления ... 624
Просмотр данных ... 626
Редактирование данных ... 627
19.3.4. Обработка ошибок ... 628
19.4. Аутентификация пользователей ... 629
19.4.1. Списки пользователей и сессии ... 629
19.4.2. Создание страницы аутентификации ... 630
19.4.3. Настройка приложения ... 631
19.5. Использование XML и XSL ... 632
19.5.1. Использование файла XML и шаблона XSL ... 633
19.5.2. Использование набора данных и шаблона XSL ... 635
19.5.3. Дополнительный компонент *TAdapterXMLBuilder* ... 636
Резюме ... 638

Приложение. Описание дискеты ... 639

Предметный указатель ... 643

Введение

При первом знакомстве с Delphi разработчики наверняка обратили внимание на то, что большинство новых возможностей и компонентов ориентированы на создание распределенных Web приложений. И, безусловно, это не дань моде, в которой фирму Borland трудно заподозрить, а насущная необходимость.

Интересно проследить эволюцию взглядов сообщества разработчиков на возможности Internet. Первоначальное представление о глобальной сети, как об очередной "технологической игрушке", имеющей, впрочем, потрясающие перспективы не только общения, но и ведения бизнеса, неизбежно породило Internet-бум. Безусловно, памятен всем ажиотаж, когда использовать Internet часто пытались не те люди и не для тех целей. Естественно, это вызвало некоторое разочарование и откат. Тем не менее, взрыв интереса к Internet как универсальному и глобальному средству бизнес-коммуникации и последовавший спад активности в этой сфере позволили выявить реальные перспективы применения Internet не только как важнейшего технологического средства, но и как системообразующего фактора проектирования современных распределенных приложений.

Всю историю развития от бизнес-приложений для мейнфреймов и до современных распределенных систем можно представить как последовательность шагов, каждый из которых делал приложения более глобальными. Безусловно, с точки зрения беспристрастного исторического исследования, такой взгляд страдает некоторой однобокостью, но в рамках данной книги представляется вполне оправданным, так как позволяет сосредоточиться на исследуемом вопросе. В этом контексте широкое использование Internet в качестве сетевой среды для распределенных бизнес-приложений является закономерным и обоснованным решением.

Популярность Internet как системообразующей платформы для распределенных приложений основана также и на крайне своевременном появлении языка XML, который позволяет унифицировать процесс представления

структуры данных. Задача описания данных XML решается при помощи другого языка — XSL (eXtended Stylesheet Language) и его модификации XSLT (XSL Transformations). И это очень важно, так как разработчики получили универсальное средство представления этих данных в море разнообразных клиентских приложений, Web-браузеров, протоколов и их расширений.

В этой книге авторы постарались подробно осветить все основные аспекты создания Web-приложений при помощи богатого инструментария среды разработки Delphi. Однако не это, а точнее не только это является главным преимуществом Delphi, как среды для создания Web-приложений. Очень важно, что Delphi позволяет осуществить полный цикл разработки сложных приложений с распределенной инфраструктурой, использующих различные технологические решения.

Действительно, разнообразие технологий и компонентов позволяет создавать приложения для любых заказчиков и выполнять большинство их пожеланий. И чтобы не быть голословными, давайте совершим небольшой экскурс в современные Internet-технологии, о которых пойдет речь в этой книге.

COM и COM+

А начнем мы с небольшого обзора технологии COM (Component Object Model) и ее дочерних технологий. В этот момент искушенный читатель может поморщиться — "Опять COM!". Однако авторы сочли необходимым обратиться к этой теме, хотя на первый взгляд вопросы применения COM и иже с ней довольно далеки от Web-приложений Де факто, многие самые современные Internet-технологии используют возможности COM как последнее звено в цепочке передачи данных — если можно так выразиться, как аналог "последней мили" сетевой инфраструктуры.

Не меньшее значение имеет и технология COM + (и ее ближайший предок — старый добрый Microsoft Transaction Server), помогающая решить проблемы масштабируемости приложений для платформы Windows 2000.

DataSnap

Рассказывая о Web-приложениях, не следует забывать о том, что Internet — безусловно замечательное и перспективное, но лишь "транспортное средство" для получения данных клиентами распределенных бизнес-приложений. Поэтому особенности архитектуры таких приложений и вопросы их взаимодействия с серверами баз данных по-прежнему актуальны.

Технология DataSnap, унаследовавшая и развившая возможности небезызвестной для профессионалов Delphi технологии MIDAS, предназначена для решения именно таких задач. Здесь речь пойдет о многозвенных распределенных приложениях и способах доступа к разнообразным источникам данных.

Необходимо отдельно сказать несколько слов о разработанной и реализованной фирмой Microsoft объектной модели доступа к данным ADO (ActiveX Data Objects), которая де факто стала стандартом в этой области. Она позволяет решить проблему распространения и инсталляции программного обеспечения промежуточного слоя доступа к данным для платформы Windows.

XML

Термин "XML" слышали все, сам язык знают многие, но далеко не все представляют горизонты практического применения этого средства. 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.

XML потому и называется "расширяемым", что дает возможность любому специалисту определить и построить свою систему тегов, описывающую его собственную предметную область.

Кроме того, XML лежит в основе многих других технологий, используемых в Delphi для создания Web-приложений.

SOAP

Со времен основания и до настоящего времени разработчики Web-приложений сталкиваются с проблемой совместимости распределенных приложений, части которых функционируют на различных программных и аппаратных платформах. В этой области предлагались различные решения. К примеру, архитектура CORBA (Common Object Request Broker Architecture) позволяет создавать полноценные мультиплатформенные решения, но требует установки специализированного клиентского программного обеспечения. Если вы хотите распространять решения на основе VisiBroker — поставляемого с Delphi брокера объектных запросов — вам следует отдельно позаботиться о приобретении прав на него. (Кстати, поддержка CORBA обеспечена в Delphi, а набор соответствующих компонентов, инструментов и утилит входит в технологию DataSnap.)

Протокол Simple Object Access Protocol (SOAP) поддерживается всеми основными платформами, представленными в Internet, не требует сложной клиентской части и обеспечивает безопасную передачу данных по Сети. Появление SOAP обусловлено хорошо известной проблемой — необходимостью совместимости приложений, работающих на разных платформах. За два года использования SOAP одобрен большинством IT-сообщества.

Набор компонентов Delphi, реализующих SOAP, позволяет конструировать полноценные клиентские и серверные части Web-приложений. Кроме этого, мы рассмотрим Microsoft SOAP Toolkit — инструмент для публикации объектов Автоматизации в качестве Web-служб и обсудим вопросы создания Web-служб.

WebSnap и WebBroker

И наконец, разработчики Delphi сумели объединить в едином технологическом решении все основные этапы создания Web-приложений. Технологии WebSnap и WebBroker упрощают и ускоряют выполнение "рутинных" операций создания пользовательского интерфейса приложений, доступа к данным, аутентификации пользователей. Кроме этого, разработчики наверняка оценят легкость встраивания в исходный код скриптов VBScript или JScript.

Широкое использование шаблонов HTML и XML является несомненным преимуществом WebSnap, так как существенно упрощает модернизацию наиболее изменчивой и подвижной части готовых приложений — пользовательского интерфейса.

Криптография и безопасность данных

Сеть Internet по сути является открытой, децентрализованной и минимально управляемой. На ранних этапах развития программирования для Internet проблема защиты данных волновала немногих, а разговоры о создании промышленных стандартов в этой сфере носили теоретический характер. В настоящее же время парадигма защиты любых важных данных, пересылаемых по Сети, не подлежит сомнению.

Поэтому в этой книге мы не могли обойти вниманием столь важную тему. Здесь рассматривается круг прикладных задач криптографии, затем вводятся понятия цифровой подписи и сертификата. Далее, на примере менеджера сертификатов, мы изучим особенности интерфейса CryptoAPI и закончим вопросами настройки Web-средств и компонентов, используемых в распределенных приложениях для криптографической защиты.

Часть I

Приложения COM и COM+

Глава 1. Механизмы COM в Delphi

Глава 2. Технология Автоматизация

Глава 3. Компоненты ActiveX

Глава 4. Технология COM+ (Microsoft Transaction Server)

Глава 1

Механизмы COM в Delphi

В основе многих технологий, рассматриваемых в этой книге, лежит *технология COM (Component Object Model, Многокомпонентная Модель Объектов)* и технологии, созданные на ее базе. Поэтому мы начнем именно с рассмотрения главных возможностей COM.

Во все времена одной из актуальнейших задач, стоящих перед специалистами, было обеспечение взаимодействия между отдельными программами. Для ее решения использовался 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 — предоставление разными программами собственных функций (служб) друг другу и правильное использование ими этих функций.

Для решения этой проблемы помимо OLE был разработан целый ряд технологий. В их основе лежит базовая технология Component Object Model (COM). Она описывает способ взаимодействия программ разных типов. Одна часть программного обеспечения предоставляет собственные службы, а другая получает к ним доступ. При этом не имеет значения, где расположены эти части — в одном процессе, в разных процессах на одном компьютере или на разных компьютерах.

Для приложений, созданных на основе спецификации COM, также не важно, какой язык программирования использовался при их разработке — если стандарт COM соблюден, 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 используется в самых различных областях разработки программного обеспечения. На основе COM созданы технологии Автоматизации (Automation), ActiveX, ActiveForm, Microsoft Transaction Server. Без применения объектов COM не смогут функционировать распределенные приложения. Независимо от того, работают ли они в локальной сети или используют Internet.

Delphi предоставляет разработчику набор инструментов для создания полноценных приложений COM.

Далее в этой главе обсуждаются 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 темы:

- объекты и интерфейсы;
- интерфейс `Iunknown`;
- библиотеки типов;
- фабрики классов;
- виды серверов.

1.1. 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 клиент может использовать службы объекта, просто вызывая его методы. Доступ к свойствам объектов осуществляется только через методы объектов.

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. 1.1 объект расположен внутри сервера — электронной таблицы. Интерфейсы обозначены маленькими кружками, связанными с объектом. Пусть интерфейс линейных функций называется `ILinear`, а интерфейс агрегатных функций — `IAggregate`.

Примечание

Согласно принятым в Delphi правилам именования, названия интерфейсов начинаются с заглавной буквы I. Напомним также, что имена классов начинаются с заглавной буквы T.

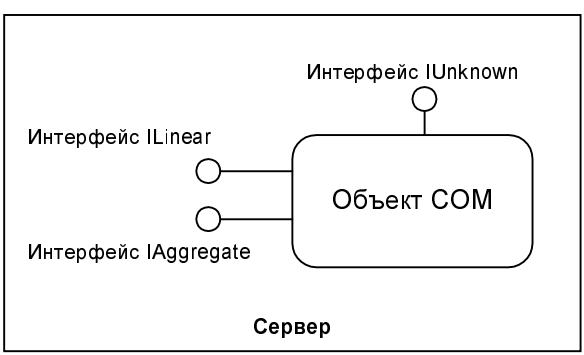


Рис. 1.1. Сервер, объект и его интерфейсы

Примечание

Согласно правилам представления объектов COM, базовый интерфейс `IUnknown` (см. рис. 1.1), имеющийся у любого объекта, обозначается как кружок, примыкающий к верхней стороне прямоугольника объекта. Остальные интерфейсы располагаются справа или слева.

На рис. 1.2 представлена схема взаимодействия клиента с объектом COM.

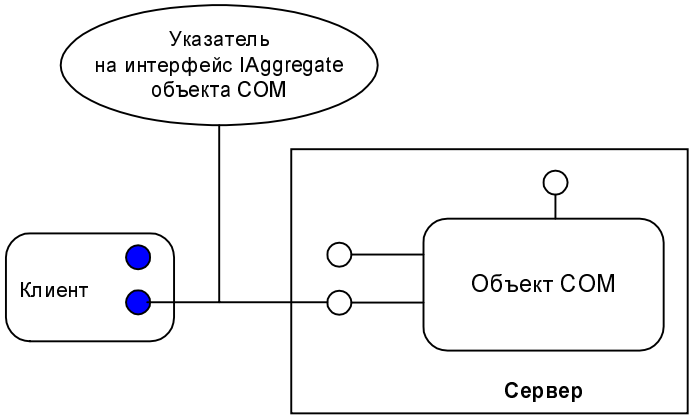


Рис. 1.2. Схема взаимодействия клиента и объекта COM

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 использование функций электронной таблицы выглядит как обычное обращение к методу объекта. Механизм обеспечения взаимодействия между удаленными элементами COM называется *маршалингом (marshalling)*.

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 используется 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 эту часть работы отвечают специальные объекты COM, которые называются *моникерами (monikers)*. Они работают скрытно от клиента. Обычно моникер создается вместе с классом.

Вполне реальным представляется одновременное обращение нескольких клиентов 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 нерассмотренным 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)*.

1.1.1 Объект

Центральным элементом COM является объект. Приложения, поддерживающие COM, включают в себя 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. Обекты создаются в соответствии с основополагающими принципами ООП, такими как: инкапсуляция, наследование и полиморфизм. Объекты COM всем этим требованиям удовлетворяют (существуют особенности наследования).

Применительно к объектам вообще понятие интерфейса объекта, как он был определен выше, не используется. 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 есть 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 используют 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.

1.1.2. Интерфейс

Если объект COM является ключевым элементом реализации COM, то интерфейсы — это 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", так как оно может вызвать нежелательные ассоциации с языком программирования, а как раз этот фактор не имеет никакого значения при взаимодействии элементов COM.).

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 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 используемого 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 есть у объекта. Для получения их перечня используется базовый интерфейс `IUnknown` *(см. разд. 1.1.3 данной главы)*, 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 используется в библиотеках типов). Его синтаксис очень похож на C++.

Теперь осталось сделать самое важное — правильно вызвать метод. Для этого в COM описана реализация интерфейса на основе стандартного двоичного формата, обеспечивающая независимость от языка программирования.

Доступный клиенту указатель интерфейса ссылается на внутренний указатель объекта и, через него, на специальную виртуальную таблицу (рис. 1.3).

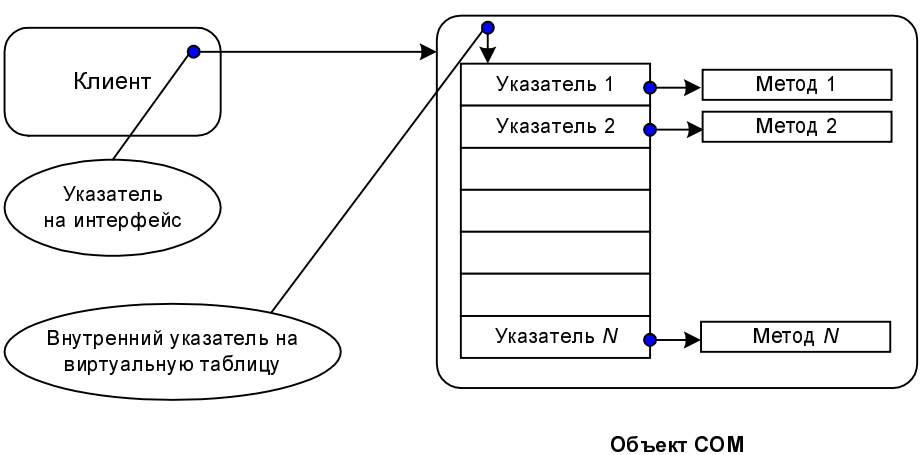


Рис. 1.3. Внутренняя структура интерфейса COM

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 его наследником.

В результате, вызов клиентом метода проходит по цепочке указателей и получает указатель на конкретный метод, а затем исполняется соответствующий программный код.

1.1.3. Интерфейс *IUnknown*

Каждый объект COM обязательно имеет интерфейс `IUnknown`. У этого интерфейса 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. Поскольку любой интерфейс является потомком `IUnknown`, то у него есть и метод `QueryInterface`. Следовательно, в общем случае не важно, какой именно интерфейс может использовать клиент. 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.

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 увеличить счетчик ссылок. Для этого используется 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.

1.1.4. Сервер

Сервер COM представляет собой приложение или динамическую библиотеку, которые могут 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 не на том компьютере, на котором выполняется клиент.

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. На 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 сервер. 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 используется механизм маршалинга и демаршалинга. *Маршалинг* реализует единый в рамках технологии COM формат упаковки параметров запроса, *демаршалинг* обеспечивает распаковку. В описанных выше реализациях серверов за выполнение этих операций отвечают заместитель и заглушка. Объекты этих типов создаются совместно с основным объектом COM с помощью IDL.

1.1.5. Библиотека COM

Для выполнения базовых функций и использования интерфейсов в операционной системе существует специальная *библиотека COM* (конкретная реализация может быть различной). Доступ к возможностям библиотеки осуществляется стандартным способом — с помощью вызова функций. Согласно спецификации, имена всех библиотечных функций начинаются с приставки "Co".

При установке приложения, поддерживающего COM, в системный реестр записывается информация обо всех реализуемых этим приложением объектах COM:

- *идентификатор класса (Class Identifier, CLSID)*, который однозначно определяет класс объекта;
- тип сервера объекта — внутренний, локальный или удаленный;
- полное имя динамической библиотеки или исполняемого файла — для локальных и внутренних серверов;
- полный сетевой адрес — для удаленных серверов.

Предположим, что клиент пытается обратиться к объекту 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` *(см. разд. 1.1.6 данной главы)*, 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. 1.4.

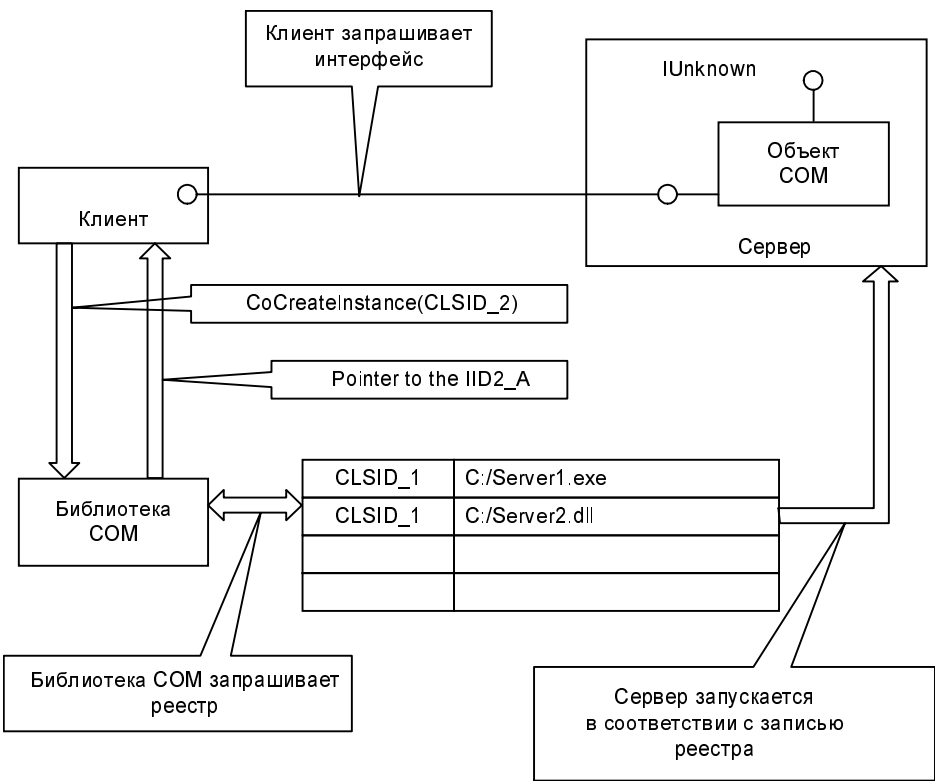


Рис. 1.4. Создание первого экземпляра объекта с помощью библиотеки COM и системного реестра

Примечание

Клиент (см. рис. 1.4) 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.

Для неявной инициализации созданного объекта (установки значений свойств) может использоваться специальный объект — моникер. Кроме того, клиент может инициализировать объект самостоятельно, применив специальные интерфейсы (`IPersistFile`, `IPersistStorage`, `IPersistStream`).

1.1.6. Фабрика класса

Для запуска экземпляра класса используется специальный объект — фабрика класса. С его помощью можно создать как один объект, так и несколько экземпляров. Для каждого класса должна существовать собственная фабрика класса.

Согласно спецификации COM фабрика класса также является объектом COM. 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 Windows API.
- `LockServer` — оставляет сервер функционировать после создания объекта.

Примечание

На самом деле функция Windows API `CoCreateInstance` при помощи переданного ей `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, используя метод `CoCreateInstance`, клиент заставляет фабрику класса создать объект.

1.1.7. Библиотека типов

Чтобы документировать интерфейсы объекта для пользователей, разработчик создает информацию о типах объекта с помощью языка IDL. Вся информация 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 интерфейсе оформляется в виде отдельного объекта внутри библиотеки.

Для создания библиотеки типов, описанной с помощью операторов IDL, используются специальные компиляторы. Доступ к библиотеке осуществляется по `CLSID` класса объекта. Кроме того, библиотека имеет собственный GUID, который сохраняется в системном реестре при регистрации объекта, для которого создана библиотека типов.

Каждая библиотека типов имеет интерфейс `ITypeLib`, который дает возможность работать с ней, как с единым объектом. Для доступа к информации об отдельном интерфейсе используется интерфейс `ITypeInfo`.

Для доступа к библиотеке по GUID применяется функция `LoadRegTypeLib`. Если клиенту известно имя файла библиотеки, то можно воспользоваться функцией `LoadTypeLib`.

1.2. Объекты COM в Delphi

Теперь рассмотрим механизм создания объектов COM в Delphi. Как говорилось ранее, объект COM должен обеспечивать возможность создания произвольного числа интерфейсов, при этом под интерфейсом понимается некоторое объединение методов, доступ к которым осуществляется с помощью указателя на интерфейс.

Реализовать такое требование напрямую в рамках стандартных подходов ООП довольно трудно. И разработчики Delphi нашли следующий выход.

Объект COM описывается обычным классом `TComObject`, который порожден непосредственно от `TObject` *(см. разд. 1.2.1 данной главы)*. Все свойства и методы, определяющие поведение объекта, описываются в его объявлении, поэтому 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 имени его класса 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 класса 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 у этого интерфейса есть доступ к библиотеке типов, то для получения полной информации об объекте достаточно знать его идентификатор класса. Этот класс используется для создания объектов с использованием библиотеки типов.

1.2.1. Класс *TComObject*

Класс `TComObject` обеспечивает выполнение базовых функций объекта, которые и делают его объектом COM. Его свойства и методы инкапсулируют функциональность (т. е. содержат программный код, описывающий выполнение операций, для реализации которых собственно и создавался класс) интерфейса `IUnknown`. Также он обеспечивает 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 интефейс `IUnknown`, который, как известно, увеличивает на единицу значение счетчика обращений и управляет всеми объектами агрегата. Объект, предоставляющий общий управляющий интефейс, 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.

Два предыдущих 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 доступно для чтения свойство

```
property RefCount: Integer;
```

которое возвращает число открытых ссылок на объект. Значение этого свойства определяется исходя из вызовов интерфейса `IUnknown` *(см. разд. 1.1.3 данной главы)*.

Методы интерфейса `IUnknown` объекта можно использовать, вызывая одноименные методы 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`.

1.2.2. Класс *TTypedComObject*

Этот класс является непосредственным потомком класса `TComObject` и применяется для создания объектов COM с использованием библиотеки типов. Класс обладает дополнительным интерфейсом `IProvideClassInfo`, который обеспечивает работу единственного дополнительного 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 используемой библиотеки типов.

1.2.3. Интерфейс *IUnknown* в Delphi

Класс `TComObject` имеет методы, которые соответствуют методам входящего в его состав 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 объявлен так:

```
IUnknown = IInterface;
```

Метод

```
function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
```

возвращает указатель на интерфейс с заданным `IID`, если он доступен.

Метод

```
function _AddRef: Integer; stdcall;
```

увеличивает счетчик ссылок на единицу.

Метод

```
function _Release: Integer; stdcall;
```

уменьшает счетчик ссылок на единицу.

Согласно спецификации COM, в Delphi любые интерфейсы являются потомками интерфейса `IUnknown`.

1.2.4. Тип глобального идентификатора

Для представления глобального уникального идентификатора GUID в Delphi в модуле System определен специальный тип:

```
TGUID = record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

Для того чтобы сгенерировать новый идентификатор, можно использовать функцию CoCreateGUID из Win API. Для преобразования идентификатора в строку используется функция

```
function GUIDToString(const ClassID: TGUID): string;
```

Для обратной операции применяется функция

```
function StringToGUID(const S: string): TGUID;
```

1.2.5. Класс *TInterfacedObject*

Специально для работы с объектами COM, содержащими 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 использования фабрики класса, обычным конструктором:

```
type
  TSomeInterfacedObject = class(TInterfacedObject, IUnknown)
  end;
...
var SomeInterfacedObject: TsomeInterfacedObject;
...
SomeInterfacedObject := TSomeInterfacedObject.Create;
```

Поскольку класс TinterfacedObject — наследник интерфейса IInterface (он же IUnknown), он содержит свойство счетчика ссылок

```
property RefCount: Integer;
```

и три базовых 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.

Примечание

Напомним, что методы класса можно вызывать без создания экземпляра объекта — с указанием имени класса, в котором они описаны. Перед описанием метода класса нужно поставить зарезервированное слово `class`.

1.3. Фабрика класса в Delphi

Создавая объекты COM в Delphi, разработчик не должен заботиться о создании фабрики класса для каждого объекта. Как будет показано дальше *(см. листинг 1.1)*, 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 на то, 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`, имеющего тип `TComObjectFactory`. Класс `TComObjectFactory` является основой для использования фабрик классов в приложениях COM в Delphi.

Если в рамках одного сервера функционирует несколько фабрик классов, используется специальный диспетчер классов COM. Он создается на базе класса `TComClassManager` и применяется 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`, используется класс `TTypedComObjectFactory`.

1.3.1. Класс *TComObjectFactory*

Этот класс инкапсулирует функции универсальной фабрики класса для объектов COM, создаваемых как экземпляры класса `TComObject`. Он обеспечивает функционирование интерфейсов: `IUnknown`, `IClassFactory`, `IClassFactory2`. Определяющим в работе фабрики класса является интерфейс `IClassFactory`.

Фабрика класса создается обычно в составе функционирующего сервера создаваемого ею объекта. Конструктор фабрики описывается в секции инициализации модуля, включающего соответствующий сервер (см. листинг 1.1).

Для создания фабрики класса используется конструктор

```
constructor Create(ComServer: TComServerObject; ComClass: TComClass;
const ClassID: TGUID; const ClassName, Description: string; Instancing:
TClassInstancing; ThreadingModel: TThreadingModel = tmSingle);
```

Параметр `ComServer` определяет сервер, в составе которого будет функционировать объект.

Параметр `ComClass` определяет тип класса, который используется для идентификации фабрики класса во время работы процедуры `GetFactoryFromClass` класса `TComClassManager` *(см. разд. 1.3.3 данной главы)*.

Параметр `ClassID` задает идентификатор класса, который создается при помощи этой фабрики класса.

Параметр `Instancing` определяет способ создания объекта.

Параметр `ThreadingModel` определяет способ взаимодействия объекта и клиента.

После создания фабрики класса основные параметры создаваемого ею объекта COM хранятся в свойствах.

Свойство

```
property ClassID: TGUID;
```

возвращает GUID класса объекта.

Свойство

```
property ClassName: string;
```

содержит имя класса объекта.

Свойство

```
property ComClass: TClass;
```

определяет тип класса, который используется 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 те же действия, что и функция `CoCreateInstance` интерфейса `IClassFactory`. 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, содержащего объект COM.

Метод

```
procedure UpdateRegistry(Register: Boolean); virtual;
```

регистрирует объект COM или удаляет его регистрацию. Для приложений это делается при первом запуске или при запуске с ключом /regserver. Метод обеспечивает сохранение в системном реестре основных параметров объекта COM и местоположения исполняемого файла.

При создании объекта COM разработчик может предусмотреть использование специального интерфейса, отвечающего 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.

Для распространения объекта COM разработчик может включить в исполняемый код лицензионную информацию и сведения о себе.

Текстовое описание объекта COM можно задать, заполнив свойство

```
property Description: string;
```

Свойство

```
property ProgID: string;
```

хранит имя разработчика, имя класса и версию.

Свойство

```
property LicString: WideString;
```

должно содержать лицензионную информацию об объекте.

Если же разработчик предусмотрел процедуру регистрации пользователей и выполнения лицензионного соглашения, то свойство

```
property SupportsLicensing: Boolean;
```

позволяет задать режим запуска с лицензированием. Для этого свойство должно иметь значение `True`.

1.3.2. Класс *TTypedComObjectFactory*

Этот класс порожден от предка `TComObjectFactory` и используется для создания фабрик классов, классы которых объявлены как `TTypedComObject`. Другими словами, 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.

1.3.3. Класс *TComClassManager*

Этот класс используется для управления фабриками класса в составе отдельного сервера COM. Переменная `ComClassManager` этого типа содержится в модуле `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 *(см. разд. 1.3.2 данной главы)*.

1.4. Сервер COM в Delphi

При создании объекта COM в модуль с его описанием автоматически добавляется модуль `ComServ`. Этот модуль описывает класс `TComServer`, содержащий свойства сервера COM, в составе которого работает соответствующий объект. Обращение к свойствам и методам этого класса позволяет получить информацию о работающих COM-объектах сервера 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;
```

Используя эту переменную, можно получать информацию от сервера.

Класс сервера применяется 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.

1.4.1. Класс *TComServer*

Этот класс объединяет свойства и методы, которые позволяют получить информацию о самом сервере и функционировании объектов в составе сервера COM. Класс является потомком абстрактного класса `TComServerObject`.

Он используется для внутренних и локальных серверов.

Если свойство

```
property IsInprocServer: Boolean;
```

имеет значение `True`, сервер является внутренним. 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.

Способ запуска файла сервера задается 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. гл. 2)*;
- `smRegServer` — сервер запускается с ключом /regserver (или запускается первый раз) для того, чтобы зарегистрироваться в системном реестре;
- `smUnregServer` — сервер запускается с ключом /unregserver для удаления регистрации.

При запуске севрера вызывается метод

```
procedure Initialize;
```

который проводит регистрацию (при первом запуске или использовании ключа /regserver) всех объектов COM, связанных с данным сервером. Например, ими могут быть главный и дочерний удаленные модули данных — специализированные объекты COM, используемые в технологии DataSnap *(см. гл. 9)*.

Имя сервера хранится в свойстве, доступном только для чтения

```
property ServerName: string;
```

Задать его можно методом

```
procedure SetServerName(const Name: string);
```

Однако если сервер использует библиотеку типов, то имя сервера определяется ею. Ссылка на библиотеку типов хранится в свойстве

```
property TypeLib: ITypeLib;
```

возвращающем 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, доступное только для чтения

```
property ObjectCount: Integer;
```

используется в процессе работы сервера. Оно возвращает общее число объектов COM, работающих с данным сервером.

1.5. Библиотека типов в Delphi

Библиотеки типов обеспечивают хранение информации об объектах, интерфейсах, функциях и т. д. Они применяются в проектах Delphi, использующих технологии, основанные на COM.

Традиционно для создания библиотек типов предназначен язык описания интерфейсов IDL (Interface Description Language). В Delphi в качестве основного варианта применяется синтаксис и операторы языка Object Pascal. Однако вы всегда можете экспортировать этот код в формат IDL, обеспечив тем самым применение собственных объектов в любых приложениях Windows.

Библиотека типов может использоваться клиентами при обращении к объекту для получения начальной информации о доступных идентификаторах, интерфейсах, методах. В принципе, эти же данные можно получить и программными средствами, используя системный реестр и возможности базовых интерфейсов COM. Однако не всегда удобно дополнять приложение довольно сложным блоком кода из-за применения небольшого объекта для вспомогательных целей.

Обычно внутри проекта используется вариант библиотеки на Object Pascal, а при распространении объектов библиотека типов экспортируется в формат IDL. Вариант библиотеки типов на Object Pascal 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 используется специальный Редактор

библиотеки типов (рис. 1.5). 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.

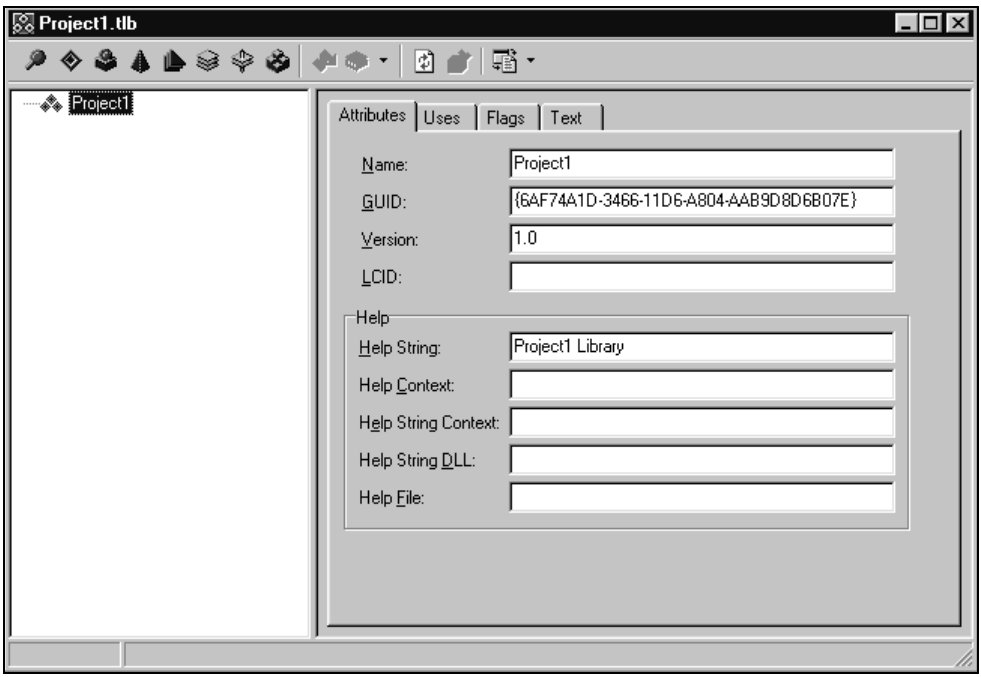


Рис. 1.5. Редактор библиотеки типов

Примечание

Разработчик может использовать или не использовать библиотеку типов в проекте по собственному желанию. Для включения библиотеки типов в проект при создании нового объекта COM необходимо установить флажок **Include Type Library** (рис. 1.6). Иначе можно воспользоваться Репозитарием объектов Delphi, в нем 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. 1.5).

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. Слева расположены кнопки новых типов, которые можно добавлять к библиотеке типов.

- (Interface) — создает новый интерфейс.
- (Dispatch) — создает новый диспинтерфейс *(более подробную информацию см. в гл. 2)*.
- (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 применяются для создания новых элементов типа. Например, для интерфейса можно создавать новые методы и переменные.
- Далее следуют две группы кнопок, обеспечивающих выполнение общих операций библиотеки.
  - Кнопка (Refresh) обновляет исходный код библиотеки, объектов и интерфейсов в других модулях.
  - Кнопка (Register) проводит регистрацию библиотеки типов в системном реестре.
  - Кнопка (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 на панели всегда присутствуют две страницы. Страница **Attributes** задает основные параметры типа. Страница **Text** может содержать текстовое описание типа.

Пример использования библиотеки типов в методах интерфейса рассматривается в разд. 1.6.1 данной главы.

1.6. 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, предоставляющего собственный интерфейс. Интерфейс будет содержать несколько методов, выполняющих простейшие математические операции.

1.6.1. Создание объекта

Для создания объекта COM в составе внутреннего сервера (динамической библиотеки) необходимо выполнить следующие действия.

На странице **ActiveX** Репозитория объектов Delphi требуется выбрать значок **ActiveX Library**. В результате будет сформирован новый проект, который мы назовем `InProcCOM`. Исходный код новой динамической библиотеки будет создан автоматически и представлен в листинге 1.1.

Листинг 1.1. 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 в Репозитории необходимо выбрать значок **COM Object** на странице **ActiveX**. После щелчка на значке открывается диалоговое окно установки начальных параметров нового объекта (рис. 1.6).

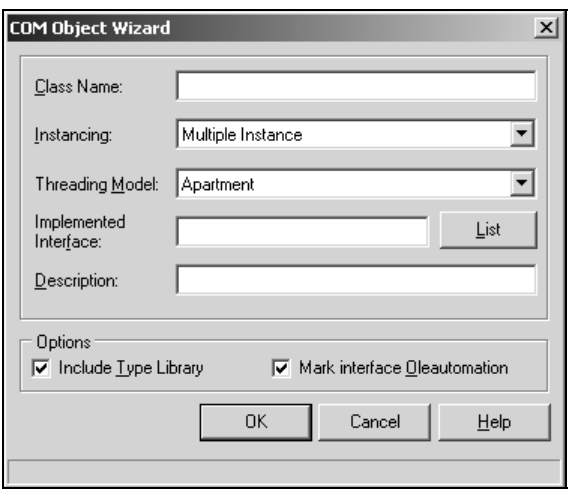


Рис. 1.6. Диалоговое окно установки начальных параметров объекта COM

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 том потоке, в котором создан и сам объект COM, при этом объект может обслуживать только одного клиента одновременно; в составе сервера в таком режиме в одно и то же время 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; используется только для технологии COM+.

Однострочный редактор **Implemented Interfaces** содержит имя интерфейса, входящего в состав создаваемого объекта. 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 в диалоговом окне **Interface Selection Wizard** (рис. 1.7), которое открывается при нажатии кнопки **List**.

Однострочный редактор **Description** содержит описание объекта.

Флажок **Include Type Library** управляет созданием библиотеки типов для объекта. Однако он будет недоступен, если вы выберете для объекта уже существующий интерфейс.

Флажок **Mark interface Oleautomation** делает объект пригодным для использования в рамках Автоматизации. Если флажок установлен, для интерфейса объекта создается диспинтерфейс *(см. гл. 2)*.

Настроенные параметры объекта `SimpleCOM` представлены в табл. 1.1.

Таблица 1.1. *Параметры объекта* `TSimpleCOM`

Параметр	Значение
Class Name	`SimpleCOM`
Instancing	`tmApartment`

Таблица 1.1 *(окончание)*

Параметр	Значение
Threading Model	`CiMultiInstance`
Implemented Interfaces	`ISimpleCOM`
Description	Simple COM Object Sample
Include Type Library	Установлен
Mark interface Oleautomation	Сброшен

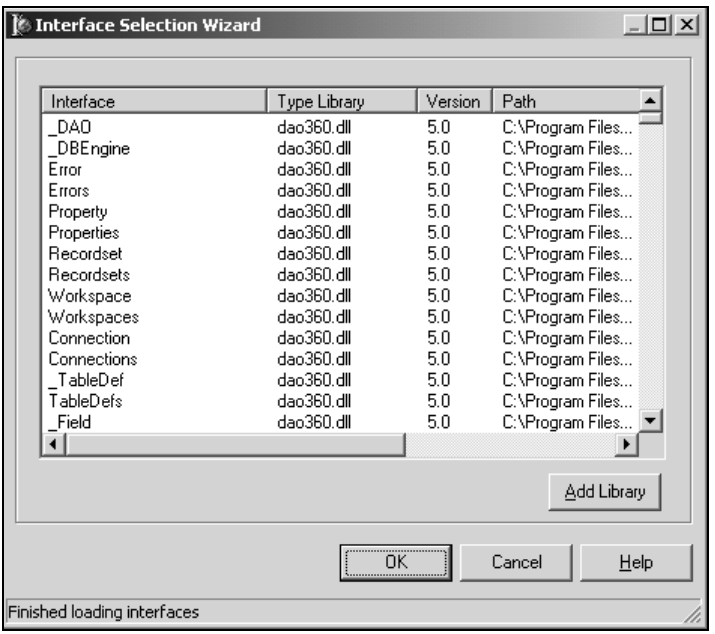


Рис. 1.7. Диалоговое окно **Interface Selection Wizard**

После установки параметров и щелчка на кнопке **OK** к проекту будет добавлен новый модуль с исходным кодом нового класса.

Код модуля объекта COM, включенный в проект сразу после создания объекта, представлен в листинге 1.2.

Листинг 1.2. 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` *(см. разд. 1.2.2 этой главы)*, так как 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, класс фабрики класса был создан автоматически.

Для обеспечения работы сервера использован модуль `ComServ` и переменная `ComServ`, указывающая на класс сервера COM. Переменная применяется в конструкторе фабрики класса.

Как видите, в модуле объекта COM описан только класс `TSimpleCOM`, который является основой функциональности объекта COM. Интерфейсы `ISimpleCOM` и `CoClass` объявлены в библиотеке типов в файле InProcCOM_TLB.PAS.

Исходный код библиотеки был создан автоматически вместе с новым объектом COM и сохранен в файле InProcCOM_TLB.PAS (листинг 1.3).

Листинг 1.3. 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 ISimpleCOM;
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` используется для работы в составе внутреннего и локального сервера, и `CreateRemote` — для работы с удаленным сервером. Обе функции возвращают указатели на интерфейс `ISimpleCOM`.

1.7. Создание методов интерфейса

Создав новый объект COM, займемся разработкой методов, которые обеспечивают его функционирование. Предположим, что объект предназначен для расчета значений некоторых 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 есть — он был создан вместе с объектом COM на предыдущем этапе.

Для добавления нового метода к существующему интерфейсу воспользуемся Редактором библиотеки типов (см. рис. 1.5). Для этого требуется выполнить следующие действия.

1. В иерархическом списке необходимо выбрать интерфейс `ISimpleCOM` и щелкнуть на кнопке **New Method** панели инструментов (рис. 1.8).

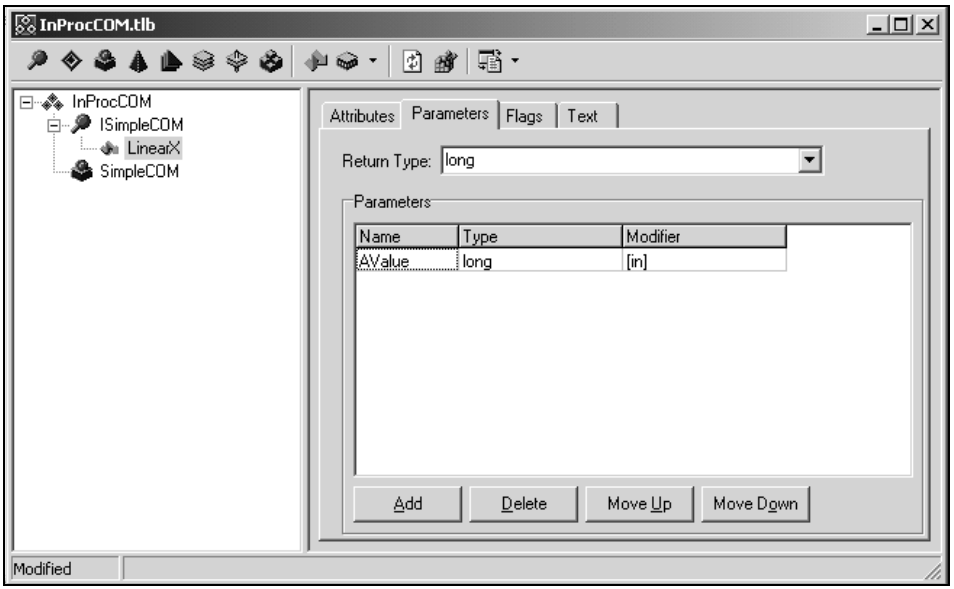


Рис. 1.8. Создание нового метода в Редакторе библиотеки типов

2. Появившийся в результате в иерархическом списке метод `Method1` переименуем в `LinearX`. Обратите внимание, 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 щелкнем на кнопке **Add** в нижней части страницы, чтобы добавить к методу первый параметр. В появившейся строке в ячейке **Name** изменим имя параметра на `AValue` (см. рис. 1.8).
5. Тип данных параметра задается в колонке **Type** (см. рис. 1.8). Раскрывающийся список колонки содержит все допустимые типы данных. Для нашего параметра выберем тип `long`.
6. Вид параметра задается в колонке **Modifier**. При щелчке на кнопке в этой колонке появляется окно **Parameter Flags** (рис. 1.9), в котором можно задать вид параметра, а также при необходимости и значение по умолчанию.

В нашем случае параметр должен быть входящим, поэтому в окне следует установить флажок **In**.

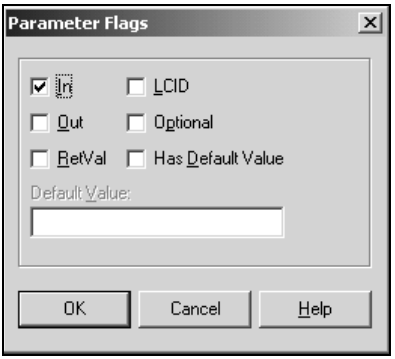


Рис. 1.9. Окно **Parameter Flags**

После выполнения этих операций объявление метода готово.

Проделаем ту же последовательность действий для метода — `SquareX`.

Исходный код этих методов в модуле объекта `SimpleCOM` создается автоматически, щелчком на кнопке **Refresh** панели инструментов Редактора библиотеки типов. Остается только ввести в них соответствующие математические функции (см. листинг 1.4).

Для создания второго интерфейса и всех необходимых методов также воспользуемся Редактором библиотеки типов. Для этого требуется выполнить следующие действия.

1. Необходимо щелкнуть на кнопке **Interface** панели инструментов и переименовать появившийся в иерархическом списке новый интерфейс, присвоив ему имя `ISimpleCOM2`.
2. На странице **Attributes** (рис. 1.10) для нового интерфейса генерируется `GUID`, а в списке **Parent Interface** следует выбрать `IUnknown` — имя интерфейса-предка.
3. На странице **Flags** необходимо сбросить флажки **Dual** и **OleAutomation**. Для вновь создаваемых интерфейсов эти флажки установлены по умолчанию, так как за основу внутреннего сервера мы взяли динамическую библиотеку ActiveX *(см. гл. 3)*.

Затем в соответствии с описанными выше действиями для нового интерфейса `ISimpleCOM2` создаются методы `Linear2X` и `CubeX`.

Теперь полученный интерфейс требуется привязать к объекту COM `TSimpleCOM`. Для этого в иерархическом списке (рис. 1.11) выбирается объект, а в правой части — страница **Implements**. На ней расположен список всех интерфейсов объекта, пока в нем только один базовый интерфейс.

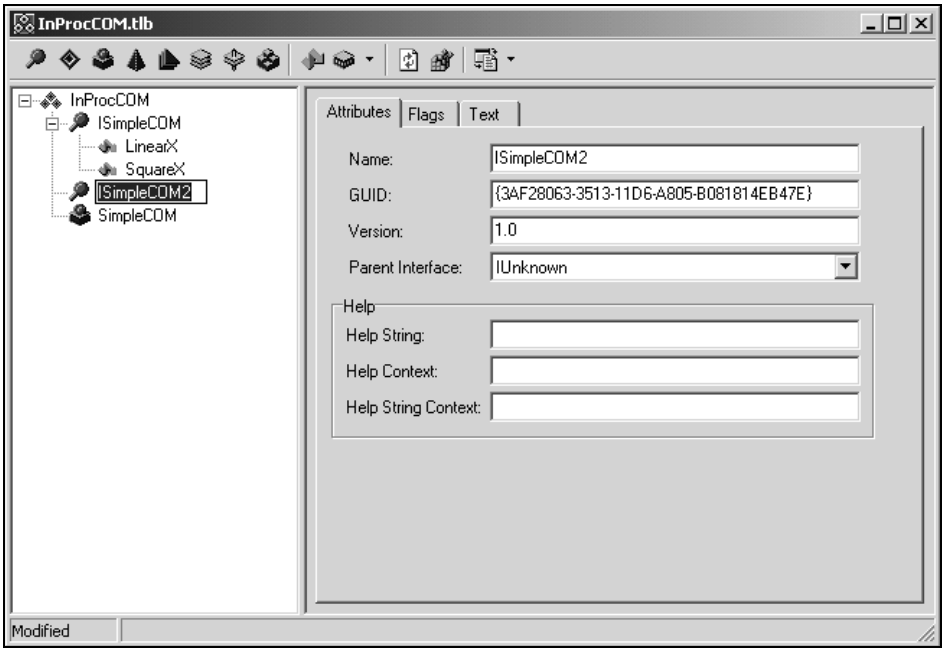


Рис. 1.10. Настройка параметров метода

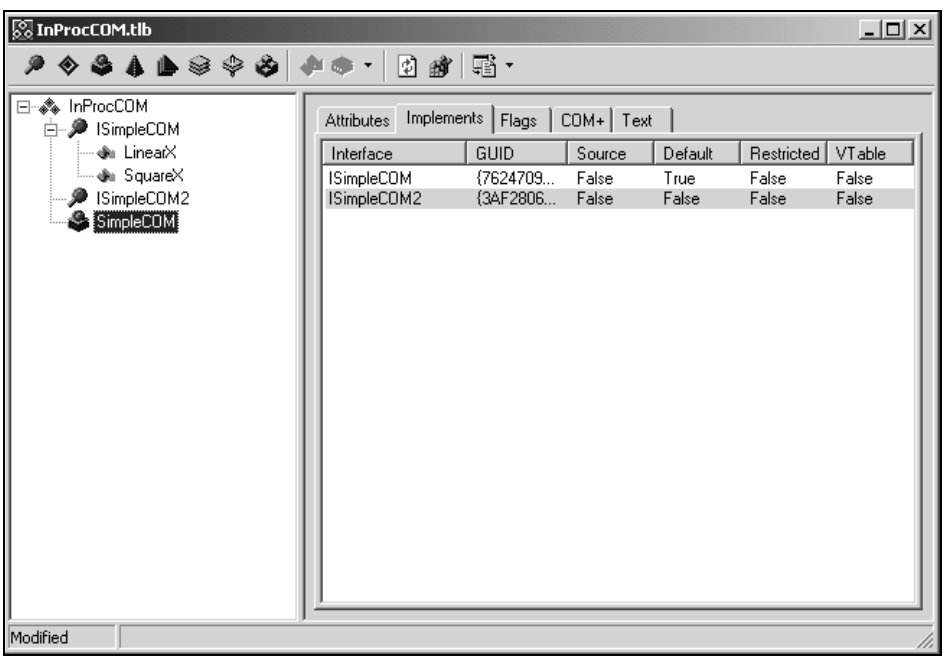


Рис. 1.11. Связывание интерфейса с объектом COM

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 (листинг 1.4). Каждый из методов выполняет элементарную арифметическую операцию.

Листинг 1.4. 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 1.5). Зато в объявлении класса `TSimpleCOM` появился второй интерфейс `ISimpleCOM2`.

Листинг 1.5. 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`.

1.7.1. 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 системный реестр.

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 использовать методы этих интерфейсов.

На этом создание объекта `SimpleCOM` закончено, можно переходить к клиентской части проекта.

1.7.2. Использование интерфейсов внутреннего сервера COM

Роль клиентского приложения сыграет простое приложение `ClientInProcCOM` с единственной формой `fmMain`. Необходимо настроить ее таким образом, чтобы можно было протестировать работу сразу двух интерфейсов созданного объекта (рис. 1.12).

Компоненты `TSpinEdit` предназначены для задания значения X, аргумента математических функций, реализованных в методах интерфейсов `ISimpleCOM` и `ISimpleCOM2`. Значения из этих визуальных компонентов будут переданы методам в качестве параметров.

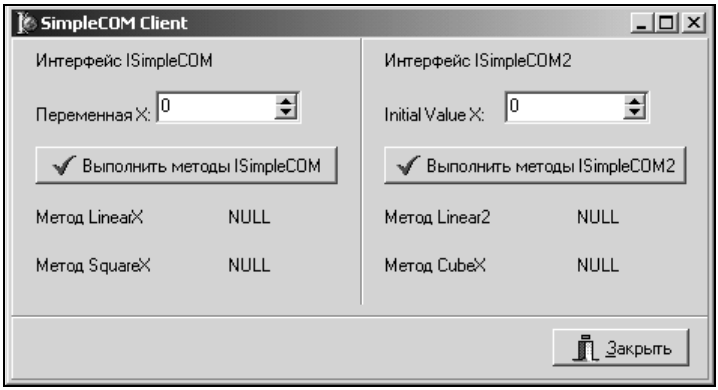


Рис. 1.12. Главная форма проекта ClientInProcCOM

Кнопки должны вызывать методы интерфейсов объекта. Для отображения результата применяются компоненты `TLabel`.

Для того чтобы клиентское приложение могло использовать интерфейсы 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;
```

В листинге 1.6 представлен исходный код клиента, обеспечивающий использование методов объекта COM.

Листинг 1.6. 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 и на различных языках программирования.

Основополагающими 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 2

Технология Автоматизация

На основе технологии COM, рассмотренной в *гл. 1*, разработаны и используются многие другие технологии. Одна из них — Автоматизация (Automation) была создана довольно давно и широко применяется в самых распространенных пользовательских приложениях ОС Windows.

Эта технология обеспечивает использование функций одних приложений из других приложений. Взаимодействие между ними осуществляется с помощью набора специально созданных интерфейсов. Основополагающим является интерфейс `IDispatch`.

Автоматизацию поддерживают такие широко распространенные программы, как Word, Excel, 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, использующих Автоматизацию, мало чем отличается от методики, рассмотренной в *гл. 1* для объектов COM. Особенности программирования в Delphi с использованием Автоматизации рассматриваются в этой главе. Среди них выделим следующие темы:

- интерфейсы автоматизации;
- объекты автоматизации;
- создание сервера автоматизации;
- создание контроллера автоматизации;
- примеры автоматизации.

2.1. Базовые понятия технологии Автоматизация

Автоматизация как технология, основанная на COM, обеспечивает предоставление функций одних приложений другим. Естественно, что это делается благодаря использованию интерфейсов. Эти интерфейсы содержатся в разнообразных объектах 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. 1.3). Такой способ доступа называется *ранним связыванием*.

В Автоматизации доступ к методам интерфейсов осуществляется как традиционным способом COM, так и при помощи специального метода, который производит поиск запрошенного метода по специальному идентификатору. Такой способ доступа называется *поздним связыванием*.

В остальном же технология Автоматизация подобна технологии COM. Рассмотрим ее основные сущности.

2.1.1. Интерфейсы 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. гл. 1)*.

Рассматриваемые здесь интерфейсы можно разделить на две группы — диспинтерфейсы и дуальные интерфейсы.

Интерфейс *IDispatch* и диспинтерфейсы

Базовым интерфейсом Автоматизации является интерфейс `IDispatch` — вполне обычный интерфейс COM, обладающий тем не менее несколькими очень важными для Автоматизации методами. Как и другие интерфейсы в COM, он реализован при помощи виртуальной таблицы указателей на его методы *(см. гл. 1)*.

Особенность же интерфейса `IDispatch` заключается в наличии специального метода `Invoke`, который обеспечивает вызов других методов. Кроме указанного метода `IDispatch` имеет еще три важных метода. Это методы `GetTypeInfoCount`, `GetTypeInfo`, `GetIDsOfNames`. Все они реализуют механизм позднего связывания.

Объект, работающий с использованием интерфейса `IDispath`, должен обязательно определить дополнительный интерфейс, в котором перечисляются методы, доступные для вызова методом `Invoke`. Такой интерфейс называется *диспетчерским интерфейсом* или *диспинтерфейсом*.

Виртуальная таблица в нем не используется. Вместо этого для доступа к методам 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 для типов данных передаваемых параметров, так при вызове методов значения всех параметров конвертируются в тип `Variant`. При возвращении результатов выполняется обратная процедура.

Рассмотрим механизм позднего связывания (рис. 2.1).

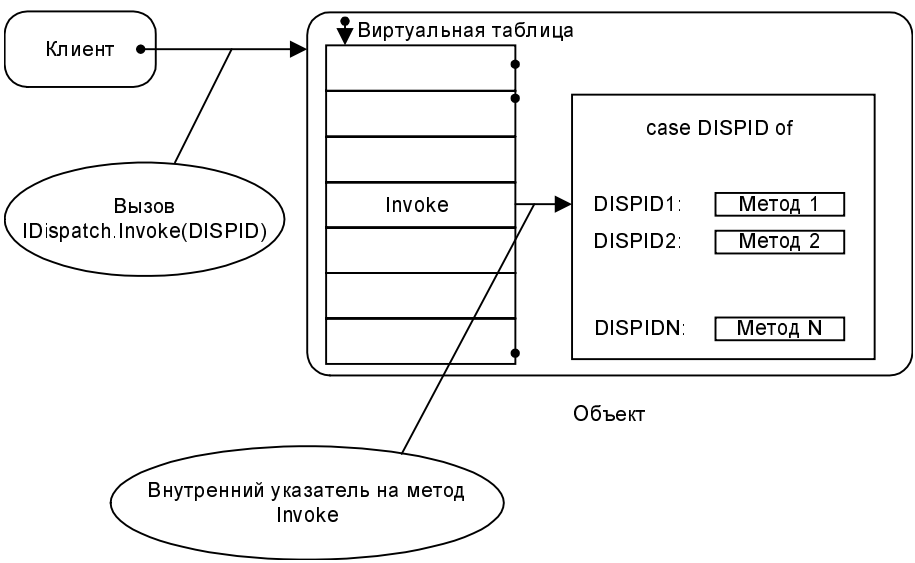


Рис. 2.1. Механизм вызова методов диспинтерфейса

Когда клиенту необходимо использовать 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 применяется тогда, когда в приложении есть несколько диспинтерфейсов. Поскольку интерфейс `IDispatch` является наследником `IUnknown`, то для вызова другого диспинтерфейса можно воспользоваться унаследованным методом `QueryInterface`.

В результате, взаимодействие между приложениями, созданными на основе Автоматизации, упрощается — ведь для успешной работы требуется всего одна виртуальная таблица для интерфейса `IDispatch`, остальные методы вызываются описанным выше способом.

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 используется еще один тип интерфейсов — *дуальные интерфейсы*. Их методы можно вызывать как с помощью `Invoke`, так и обращаясь к виртуальной таблице.

Дуальный интерфейс обязательно должен быть наследником `IDispatch`. Его виртуальная таблица включает ссылки на три метода `IUnknown`, четыре метода `IDispatch` и методы соответствующего диспинтерфейса.

2.1.2. Библиотека типов

Поскольку при вызове методов диспинтерфейса клиент должен обладать информацией о диспетчерских идентификаторах вызываемых методов, при создании интерфейсов Автоматизации значительную роль играет Библиотека типов *(см. гл. 1)*. Предоставленные библиотекой типов идентификаторы методов, а также информация о параметрах методов и самих интерфейсах используются клиентом при вызове метода `Invoke` интерфейса `IDispatch`.

2.1.3. Маршалинг интерфейсов Автоматизации

Для обеспечения работы локальных и удаленных серверов Автоматизации также применяется механизм маршалинга и демаршалинга. Маршалинг отвечает за упаковку параметров запроса, демаршалинг обеспечивает распаковку *(см. гл. 1)*.

Как и любой интерфейс с виртуальной таблицей методов, интерфейс IDispatch и его потомки используют для маршалинга и демаршалинга пару вспомогательных объектов — заместитель и заглушку *(см. гл. 1)*. 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 `Variant`. Таким образом, маршалинг диспинтефрейсов имеет особенность — для передачи разнотипных параметров методом `Invoke` применяется конвертация параметров в тип `Variant` и обратно. Поэтому методы интерфейсов Автоматизации используют ограниченный набор типов данных, пригодных для подобного преобразования.

Именно потому, что при маршалинге диспинтерфейсов не требуются заместитель и заглушка, диспинтерфейсы поддерживают позднее связывание.

2.1.4. Объект Автоматизации

В рамках технологии Автоматизация (так же как и в COM) все методы инкапсулированы в *объекте Автоматизации*, который может содержать один или несколько 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.

2.1.5. Сервер Автоматизации

Приложение, предоставляющее свои функции при помощи интерфейса `IDispatch`, называется *сервером Автоматизации*. Сервер обязательно должен включать *объект автоматизации* — обычный объект COM, имеющий интерфейс `IDispatch` и диспинтерфейсы.

Сервер Автоматизации — это исполняемый файл, который представляет собой приложение или динамическую библиотеку и содержит один или несколько объектов одного или разных классов. Как и серверы COM, серверы Автоматизации могут быть трех типов: внутренний сервер, локальный сервер и удаленный сервер *(см. гл. 1)*.

2.1.6. Контроллер Автоматизации

Любой стандартный клиент COM, который обращается к серверу Автоматизации через `IDispatch`, называется *контроллером Автоматизации*.

2.2. Реализация Автоматизации в Delphi

Теперь рассмотрим, 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 используется информация о типах. Библиотека типов Автоматизации создается так же, как и для обычных приложений COM *(см. гл. 1)*.

2.2.1. Интерфейсы автоматизации

Все интерфейсы Delphi, используемые в 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 Автоматизация 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): HRe-
sult; 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 методов, 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;
```

используется 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 указывает, вызывается обычный метод либо метод чтения или записи свойства. Параметр Params хранит указатель на массив типа TDispParams, содержащий параметры вызываемого метода. Параметр VarResult возвращает значение, переданное вызываемым методом.

Если при вызове метода возникла ошибка, параметр `ExcepInfo` сохраняет информацию об этом. 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` хранится число интефейсов 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` хранит указатель на структуру `ITypeInfo`, содержащую информацию о типе.

Метод

```
function GetIDsOfNames(const IID: TGUID; Names: Pointer; NameCount, Lo-
caleID: Integer; DispIDs: Pointer): Hresult; stdcall;
```

по заданным в массиве `Names` именам методов находит их диспетчерские идентификаторы и возвращает в параметре `DispIDs` указатель на массив, их содержащий. Размерность массива `Names` определяется параметром `NameCount`.

2.2.2. 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 используется 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. У параметров и возвращаемых результатов методов

могут быть только следующие типы: `Byte`, `Currency`, `Real`, `Double`, `Real48`, `Integer`, `Single`, `Smallint`, `AnsiString`, `ShortString`, `TDateTime`, `Variant`, `OleVariant`, `WordBool`.

Для обращения к методам и свойствам интерфейсов диспетчеризации используются методы `GetIDsOfNames` и `Invoke`. Интерфейсы диспетчеризации работают на основе позднего связывания.

2.2.3. Дуальные интерфейсы

Используемые в приложениях Автоматизации Delphi дуальные интерфейсы создаются на базе `IDispatch`. При объявлении их методов обязательно следует использовать директиву `safecall`. Параметры и возвращаемые результаты методов должны иметь только следующие типы: `Byte`, `Currency`, `Real`, `Double`, `Real48`, `Integer`, `Single`, `Smallint`, `AnsiString`, `ShortString`, `TDateTime`, `Variant`, `OleVariant`, `WordBool`.

Дуальные интерфейсы обеспечивают как связывание на этапе компиляции приложения (виртуальная таблица), так и связывание во время выполнения, или позднее связывание, используемое в Автоматизации (метод `Invoke`).

Первые три метода дуального интерфейса наследуются от `IUnknown`, следующие четыре метода — от `IDispatch`, затем идут собственные методы интерфейса.

2.2.4. Объект Автоматизации

Возможности объекта Автоматизации в Delphi инкапсулирует класс `TAutoObject`. Он — наследник классов `TCOMObject` и `TTypedCOMObject` (рис. 2.2).

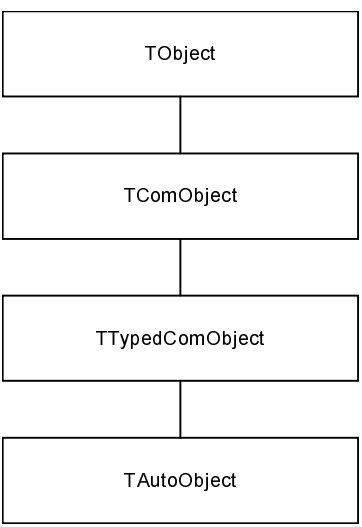


Рис. 2.2. Иерархия предков класса `TAutoObject`

Рассмотрим его основные свойства и методы, а затем пример создания объекта Автоматизации.

Класс *TAutoObject*

Класс объекта Автоматизации `TAutoObject` инкапсулирует интерфейс `IDispatch`, следовательно, у него есть четыре метода, аналогичные методам интерфейса. Это методы `Invoke`, `GetTypeInfo`, `GetTypeInfoCount`, `GetIDsOfNames`. Кроме того, как наследник класса `TComObject` класс объекта Автоматизации обладает и тремя методами интерфейса `IUnknown` *(см. гл. 1)*.

Для создания экземпляра объекта Автоматизации, также как и для любого объекта COM, необходима фабрика класса *(см. разд. 2.2.5 данной главы)*. В созданном объекте на использованную 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 задать там необходимые при инициализации объекта действия.

Для создания нового объекта Автоматизации используйте Репозиторий Delphi. При щелчке на значке **Automation Object** страницы **ActiveX** появляется окно Мастера создания объекта Автоматизации (рис. 2.3).

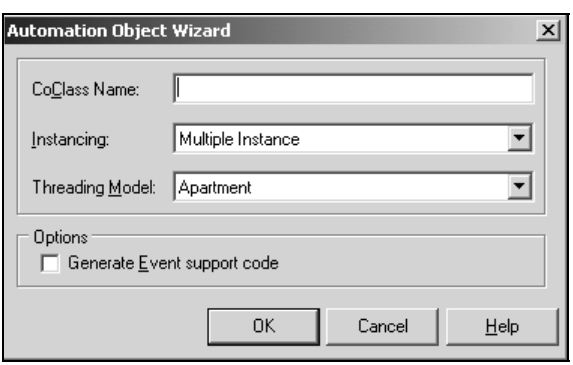


Рис. 2.3. Мастер создания объекта Автоматизации

Поле **CoClass Name** используется для задания имени нового объекта.

Список **Instancing** определяет способ запуска сервера. Список **Threading Model** позволяет выбрать способ взаимодействия объекта и контроллеров. Возможные значения этих параметров подробно описаны в *гл. 1*.

Взведенный флажок **Generate event support code** добавляет к коду создаваемого объекта Автоматизации дополнительный интерфейс, который обеспечивает управление событиями объекта на стороне сервера *(см. разд. 2.2.6 этой главы)*.

Одновременно с объектом Автоматизации создается одноименный `CoClass` и два интерфейса: одноименный с объектом интерфейс, наследник 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;
```

наследник класса `TautoObject`, который содержит 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. В дальнейшем при добавлении методов к основному интерфейсу, эти же методы появятся и в диспинтерфейсе 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 взвести флажок **Generate event support code**, то к новому объекту будет добавлен механизм обработки событий. Этот механизм 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, который автоматически генерируется при установке флажка **Generate event support code** (листинг 2.1).

Листинг 2.1. 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`.

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` (листинг 2.2).

Листинг 2.2. 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.
```

Программный код, представленный в листингах 2.1 и 2.2, 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, сопровождая их 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`:

```
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 использовать аналогичные события класса обработки событий.

```
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;
```

2.2.5. Фабрика класса

Фабрика класса для объекта Автоматизации создается на основе класса `TAutoObjectFactory`. Фабрика класса Автоматизации — наследник стандартной фабрики класса COM.

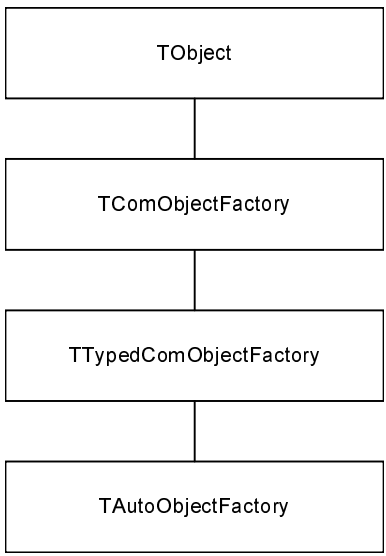


Рис. 2.4. Иерархия классов-предков фабрики класса Автоматизации

Фабрика класса Автоматизации создается конструктором

```
constructor Create(ComServer: TComServerObject; AutoClass: TAutoClass;
const ClassID: TGUID; Instancing: TClassInstancing;ThreadingModel:
TThreadingModel = tmSingle);
```

Как видно из объявления, список его параметров совпадает со списком параметров конструктора фабрики класса COM *(см. гл. 1)*.

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 о диспинтерфейсе, заданном парметром `GUID`.

Информация о типе диспинтерфейса, созданного фабрикой объекта Автоматизации, содержится в свойстве

```
property DispTypeInfo: ITypeInfo;
```

Для поддержки интерфейса обработки событий, рассмотренного ранее, используется 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 требуется использовать внутри приложения, то можно применить класс `TAutoIntfObject`, который для создания экземпляра класса не требует фабрики класса. Он поддерживает интерфейсы `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, представленной в параметре `TypeLib`.

После создания глобальный идентификатор диспинтерфейса объекта Автоматизации хранится в свойстве

```
property DispIID: TGUID;
```

Информацию о типе диспинтерфейса содержит свойство

```
property DispTypeInfo: ITypeInfo;
```

А структура `TinterfaceEntry`, описанная ранее, хранится в свойстве

```
property DispIntfEntry: PInterfaceEntry;
```

2.2.6. 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 используется мастер, открывающийся при щелчке на значке **Automation Object** страницы **ActiveX** в Репозитории Delphi.

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 Delphi **Register ActiveX Server** из пункта меню **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.

2.2.7. Контроллер Автоматизации

Контроллер Автоматизации управляет работой сервера, используя методы принадлежащих серверу интерфейсов. Информация об интерфейсах обычно распространяется в виде библиотек типов *(более подробную информацию о языках библиотек типов см. в гл. 1)*. Любое приложение, имеющее доступ к соответствующей библиотеке типов, может стать контроллером Автоматизации.

Для включения библиотеки типов в состав проекта можно использовать команду **Import Type Library** из пункта меню **Project**. Если библиотека создана в Delphi, можно просто включить имя ее файла в секцию `uses` модуля проекта.

В списке диалогового окна импорта библиотеки типов (рис. 2.5) содержатся все зарегистрированные в системе серверы Автоматизации. Для любого из них можно импортировать библиотеку типов. Список редактируется кнопками **Add** и **Remove**.

После этого в контроллере необходимо создать экземпляр класса-оболочки интерфейса (`CoClass`) и применить в нужных местах нужные методы сервера Автоматизации.

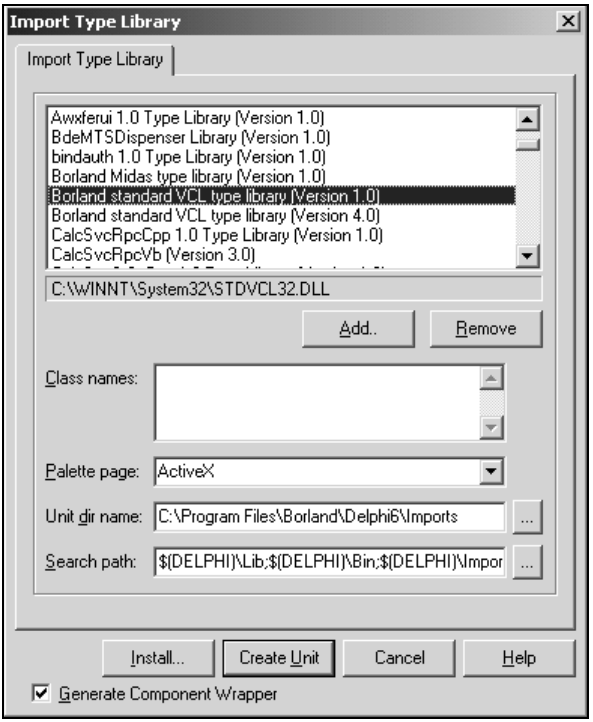


Рис. 2.5. Диалоговое окно импорта библиотеки типов

Если библиотека типов недоступна, разработчик должен поступить следующим образом.

1. Сначала на основе переменной типа `OleValiant` в приложении-контроллере создается объект Автоматизации. Для этого используется 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`. Приведенная функция используется для внутренних и внешних серверов. Естественно, сервер, для которого создается объект, должен быть зарегистрирован.

2. После создания объекта Автоматизации методы его интерфейсов можно использовать в приложении-контроллере. Если при первом обращении сервер не работает, то он автоматически запускается операционной системой.

2.3. 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.

2.3.1. 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 используется Репозиторий объектов. На странице **ActiveX** необходимо выбрать значок **Automation Object** (рис. 2.6).

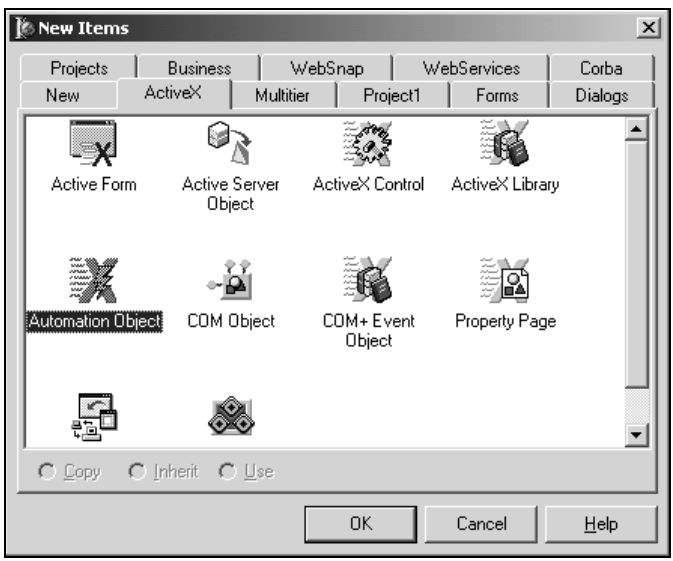


Рис. 2.6. Репозиторий Delphi, страница ActiveX

При создании нового объекта Автоматизации появляется диалоговое окно **Automation Object Wizard** (см. рис. 2.3). Подробно процесс создания объекта рассматривался ранее в этой главе. Полностью в соответствии с этим описанием был создан объект `TSimple` (табл. 2.1.).

Таблица 2.1. *Параметры объекта* `TSimple`

Параметр	Значение
CoClass Name	`Simple`
Instancing	`tmApartment`
Threading Model	`ciMultiInstance`
Generate Event support code	Установлен

Создание и настройка свойств и методов интерфейсов объекта осуществляется в Редакторе библиотеки типов. В нем были созданы объявления для двух методов: `Method1` и `Method2`. *Более подробную информацию о создании методов интерфейсов см. в гл. 1.*

Обратите внимание (листинг 2.3), интерфейс `ISimple` объекта Автоматизации полностью соответствует требованиям, предъявляемым к дуальному интерфейсу. Наряду с ним автоматически создается интерфейс диспетчеризации `ISimpleDisp`. В остальном исходный код библиотеки типов соответствует стандартному объекту COM.

Листинг 2.3. 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 2.4).

Листинг 2.4. 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 диалоговое окно параметров запуска (команда **Parameters** меню **Run**).

2.3.2. 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 предла-

гаемом нами примере, так как сервер и контроллер расположены в одной группе проектов.

Форма контроллера включает лишь две кнопки, которые будут использованы для вызова методов сервера Автоматизации (листинг 2.5).

Листинг 2.5. 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 класса-оболочки 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 3

Компоненты ActiveX

В предыдущих главах рассматривались вопросы создания объектов для технологий COM и Автоматизация. Эти объекты предоставляют клиентам наборы стандартных (определенных в рамках соответствующей спецификации) и собственных интерфейсов. Методы и свойства этих интерфейсов могут использоваться другими приложениями. Тем самым реализуется программный интерфейс COM для таких объектов.

Теперь настало время распространить 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 для методов, обеспечивающих взаимодействие с пользователем и отображение элемента управления на экране?

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 выполнять самые разнообразные дополнительные функции визуального интерфейса приложений, предоставляя пользователям дополнительные удобства и возможности.

В составе Палитры компонентов Delphi есть страница **ActiveX**, на которой представлены четыре компонента, являющиеся элементами управления ActiveX. По терминологии, принятой в Delphi, все элементы управления, доступные программисту в среде разработки, являются компонентами. При работе в Delphi элементы управления AciveX не отличаются от обычных компонентов, поэтому в дальнейшем изложении мы будем называть их *компонентами ActiveX*.

А теперь обратимся к истории вопроса.

Компонентное программирование начиналось с Microsoft Visual Basic. Именно в этой среде впервые были применены 16-разрядные модули расширения *VBX (Visual Basic eXtension)*. Наличие такой возможности 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 завоевал еще большую популярность, 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 первые элементы управления (тогда они еще назывались элементы управления OLE) в соответствии со спецификацией обязаны были поддерживать слишком много стандартных интерфейсов. И это притом, 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, а в результате элементы управления стали гораздо более компактными и дали толчок развитию ряда смежных технологий. 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 использовать всю полноту их возможностей. С точки зрения модели объектов COM элемент управления ActiveX — это сервер, поддерживающий технологию Автоматизация, реализованный в виде динамической библиотеки, исполняемый в адресном пространстве вашего приложения и допускающий визуальное редактирование.

В этой главе мы ответим на следующие вопросы.

- Как работают элементы управления ActiveX?
- Как включить в состав Delphi и использовать готовые элементы управления, созданные сторонними разработчиками?
- Как стандартные компоненты Delphi превратить в компоненты ActiveX?
- Как преобразовать форму в форму ActiveX?

3.1. Как работают элементы управления ActiveX

Элемент управления ActiveX является объектом COM, функционирующим чаще всего в составе внутреннего сервера *(см. гл. 1)*. Конечно же, технология ActiveX налагает на элементы управления ряд дополнительных ограничений,

но все они обусловлены существующими механизмами загрузки и функционирования элементов управления.

Являясь частью визуального интерфейса приложения, 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.

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 сочтет необходимым сделать свои элементы управления недоступными, то и наш элемент должен среагировать на этот сигнал приложения.

С формальной точки зрения элемент управления ActiveX обязан поддерживать лишь интерфейс `IUnknown`. Однако для реализации основных функций, присущих элементам управления ActiveX, он должен реализовывать некоторое подмножество из набора базовых интерфейсов ActiveX. Их перечень представлен в табл. 3.1.

Таблица 3.1. *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`	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`	Предоставляет два метода для установки режима безопасного использования элемента управления. Если установить этот режим, то, например, при загрузке через Internet браузер будет получать информацию о том, что данный элемент безопасен для используемых данных и для окружения, в котором он загружается
`IPersistPropertyBag`	Предоставляет методы для сохранения и загрузки индивидуальных свойств элемента управления в хранилище
`IPersistStreamInit`	Обеспечивает инициализацию хранилища, предназначенного для сохранения значений свойств элемента управления на основе использования потоков внутри хранилища

Таблица 3.1 *(окончание)*

Интерфейс	Описание
`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 в нем других элементов управления AcitveX
`ISpecifyPropertyPages`	Содержит метод, возвращающий информацию о наличии и поддержке данным элементом управления страниц свойств
`IViewObject`	Обеспечивает взаимодействие элемента управления и контейнера для передачи сообщения контейнеру об изменениях во внешнем виде элемента управления
`IViewObject2`	Расширение интерфейса IViewObject. Предоставляет информацию о размерах области прорисовки элемента управления

Доступ к методам интерфейсов осуществляется методом `Invoke` интерфейса `Idispatch` (рис. 3.1).

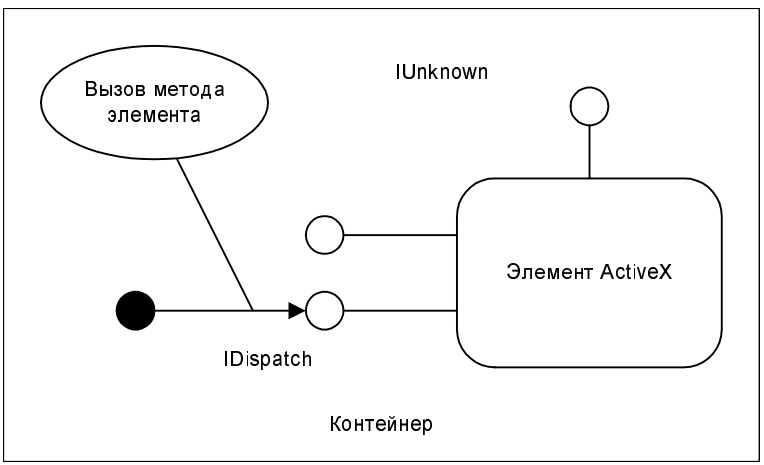


Рис. 3.1. Диспинтерфейсы, используемые для доступа к элементу управления ActiveX

Итак, для элементов управления ActiveX можно выделить следующие основные задачи:

- регистрация элемента управления;
- реализация пользовательского интерфейса и визуализация;
- предоставление своих методов и свойств приложению;
- оповещение приложения о возникающих событиях;
- получение информации о свойствах приложения.

Технология ActiveX наилучшим образом справляется с такими задачами: она определяет и реализует стандарты поведения компонентов пользовательского интерфейса приложения.

Давайте посмотрим, как решаются эти задачи.

3.1.1. Контейнеры и регистрация элементов управления ActiveX

Перечисленные выше 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 использовать, называется *контейнером элементов управления*. Пользователи работают со многими программами и даже не подозревают, что эти приложения 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, то оно уже является контейнером. Перечень базовых интерфейсов для контейнеров ActiveX представлен в табл. 3.2.

Таблица 3.2. *Базовые интерфейсы элементов управления ActiveX*

Интерфейс	Описание
`IAdviseSink`	Дает возможность контейнеру отслеживать изменения в элементах управления, связанных документах и т. д.
`IOleClientSite`	Предоставляет присоединенным объектам информацию о ресурсах контейнера

Таблица 3.2 *(окончание)*

Интерфейс	Описание
`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 диалоговых окнах 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. При установке элементы управления ActiveX заносят сведения о себе в системный реестр. Подробнее процедура регистрации рассматривается ниже.

3.1.2. Предоставление методов

Все стандартные интерфейсы контейнеров и компонентов ActiveX являются интерфейсами диспетчеризации и происходят от `IDispatch`. Соответственно любые вызовы методов элементов управления из контейнера выполняются методом Invoke интерфейса `IDispatch` *(см. гл. 2)*. Таким образом, контейнер использует только интерфейс `IDispatch` и ничего кроме него (см. рис. 3.1).

Для получения информации о методах диспинтерфейсов контейнер обращается к библиотеке типов, предоставляемой элементом управления.

3.1.3. События

Реагируя на действия пользователя, элемент управления ActiveX посылает контейнеру информацию о событиях, предоставляя тому возможность реагировать на них.

Как же работает передача сообщений в ActiveX?

При возникновении события элемент управления просто-напросто вызывает необходимый метод контейнера. В этом случае контейнер предоставляет свои интерфейсы, а элемент действует как клиент. 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 случае жизни, для любых типов компонентов. Как вы понимаете, такой экстенсивный способ реализации был бы крайне неэффективен.

Следовательно, 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.

3.1.4. Свойства

Поскольку элементы управления ActiveX используют диспинтерфейсы, они могут использовать свойства интерфейсов *(см. гл. 2)*.

Для свойства могут быть определены два метода — метод чтения и метод записи. Эти методы, так же как и любые другие методы диспинтерфейсов, вызываются методом `IDispatch.Invoke`.

3.1.5. Страницы свойств

Для стандартизации визуального представления свойств элементов управления ActiveX в них реализованы *страницы свойств*. Страницы свойств 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 диалогового окна страниц свойств.

Объект-рамка и объект-вкладка взаимодействуют друг с другом при помощи пары интерфейсов: первый предоставляет интерфейс `IPropertyPageSite`, а второй — интерфейс `IPropertyPage`.

Если пользователь изменил значение свойства в диалоговом окне страниц свойств, соответствующий объект-вкладка обеспечивает передачу нового значения элементу управления.

3.1.6. Лицензирование

Для элементов управления ActiveX, также 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 из 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 использование этого элемента на данной машине при помощи методов интерфейса `IClassFactory2`.

Метод `GetLicInfo` проверяет наличие глобальной лицензии. Если она есть, 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, используемый в вышеописанном методе, контейнер должен вызвать еще один метод — `RequestLicKey`, который возвращает искомый ключ (если он, конечно, есть).

3.2. Реализация компонентов ActiveX в Delphi

Компоненты ActiveX в Delphi используют те же механизмы, что и обычные объекты COM. Классы компонентов наследуют свойства и методы класса-предка и интерфейса-предка. Компоненты ActiveX предоставляют клиентам свои интерфейсы, которые происходят от `IDispatch`. Для создания экземпляров компонентов ActiveX используются 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.

Иерархия классов-предков компонентов ActiveX (рис. 3.2) показывает, что технология ActiveX базируется на функциональности родительских технологий — COM и Автоматизация.

Процесс создания компонентов ActiveX в Delphi существенно облегчен за счет использования мастера, генерирующего весь необходимый исходный код для нового компонента, его фабрики класса и библиотеки типов.

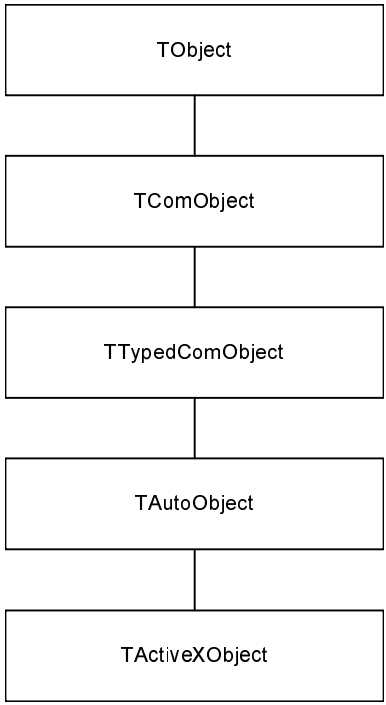


Рис. 3.2. Иерархия классов компонентов ActiveX в Delphi

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`.

3.2.1. Класс компонента ActiveX

Итак, непосредственным предком любого компонента ActiveX в Delphi является класс `TActiveXControl`. Происходя от базовых классов объектов COM и Автоматизации, он предоставляет своим потомкам механизм использования диспинтерфейсов, необходимых для работы элементов управления ActiveX.

Кроме этого, он содержит все базовые интерфейсы ActiveX, перечисленные в табл. 3.1.

Применять конструктор класса нет необходимости, так как для создания экземпляра компонента ActiveX используется фабрика класса. Но при создании компонента будет вызван метод

```
procedure Initialize; override;
```

который вы можете перекрыть и применить для установки начальных значений и инициализации вспомогательных объектов в собственном элементе управления ActiveX.

Класс предоставляет для использования готовый механизм уведомления о действиях с компонентом. Если необходимо изменить значение какого-либо свойства в ответ на действия пользователя, следует удостовериться в том, что это свойство можно менять. Для этого применяется один из вариантов перегружаемого метода

```
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 доступно для записи, функция возвращает `True`.

При изменении значения какого-либо свойства компонента вы можете вызвать один из вариантов перегружаемого метода

```
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, так как 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, содержащего ваш компонент ActiveX. Для этого применяется свойство

```
property Control: TWinControl;
```

Для получения указателей на необходимые интерфейсы можно использовать 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`.

3.2.2. Фабрика класса компонента ActiveX

Для создания экземпляров компонентов ActiveX в Delphi используется фабрика класса. Ее функциональность унаследована от аналогичной фабрики технологии Автоматизация. Фабрика класса для компонентов ActiveX входит в состав класса `TActiveXControlFactory`.

3.2.3. Среда разработки Delphi как контейнер ActiveX

Среда разработки Delphi является контейнером ActiveX.

При создании с помощью мастера ActiveX Control Wizard нового компонента ActiveX в Delphi для него автоматически создается специальный класс-оболочка. Назначение этого класса — предоставлять 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 использовать методы этого класса напрямую.

3.2.4. Регистрация компонентов ActiveX

Для регистрации созданных в Delphi элементов управления ActiveX можно использовать возможности операционной системы или среды разработки.

В операционной системе вы можете использовать системную утилиту 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 пункта меню **Run**.

Для отмены регистрации элемента управления применяется команда **Unregister ActiveX Server** из пункта меню **Run**.

3.3. Использование готовых компонентов ActiveX

С Delphi поставляется ряд компонентов ActiveX, созданных сторонними разработчиками (`ChartFX`, `VSSpell` и другие), однако вам наверняка придется устанавливать и другие элементы управления, расширяя возможности среды. Опишем этот путь и те преграды, которые могут на нем встретиться.

3.3.1. Инсталляция готовых элементов управления AciveX

Первым делом следует убедиться, что информация о нужных вам элементах ActiveX есть в реестре. Для этого нужно выбрать пункт меню **Component/ Import ActiveX Control**. В появившемся диалоговом окне (рис. 3.3) в списке в верхней части окна представлена информация обо всех элементах ActiveX, зарегистрированных в системе. В строке под списком выбора показано, в каких файлах они находятся.

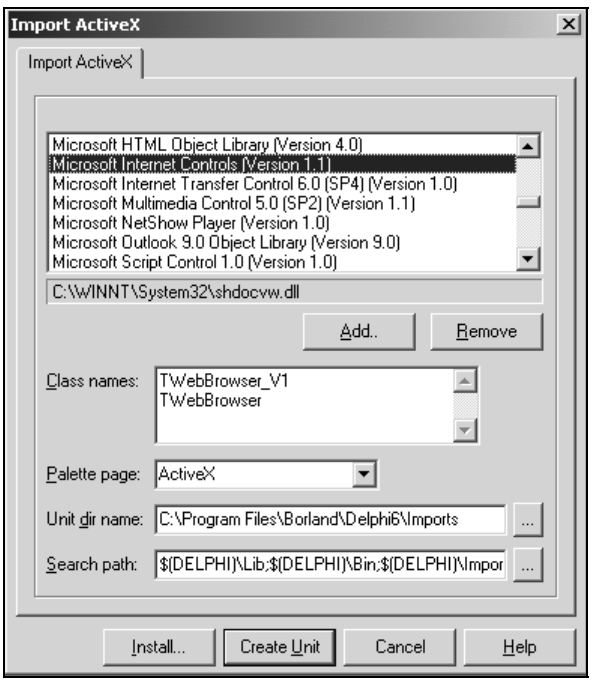


Рис. 3.3. Диалоговое окно импорта элементов управления ActiveX

Как правило, все элементы ActiveX, распространяемые в составе крупных программных продуктов, регистрируются автоматически при инсталляции. Если вы скопировали файл из Internet или взяли у знакомого, то зарегистрировать его можно, по крайней мере, тремя способами:

- нажатием кнопки **Add** в диалоговом окне **Import ActiveX**;
- вызовом системной утилиты regsvr32.exe, входящей в состав операционной системы;
- вызовом утилиты tregsvr, которая поставляется с Delphi в исходных текстах (папка ..\Demos\ActiveX\TRegSvr).

Отмена регистрации элемента управления осуществляется нажатием кнопки **Remove**.

После того как вы остановились на одном из элементов, в поле **Class Names** будут перечислены все классы, содержащиеся в данном файле. В списке **Palette Page** вам предлагается выбрать "посадочную площадку" в Delphi для выбранного элемента — страницу Палитры компонентов.

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 Object Pascal. Его имя складывается из имени ActiveX и строки "_TLB.PAS". Нажатие кнопки **Create Unit** только создает этот файл (для ознакомления и тестирования), а нажатие кнопки **Install** продолжает процедуру установки.

Устанавливаемый элемент управления должен быть помещен в один из *пакетов (packages)*. Можно выбрать один из существующих файлов с расширением DPK или создать новый. 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. Кроме того, новый пакет понадобится, если вы уже определились с выбором и вам не нужна лишняя трата памяти.

3.3.2. Деинсталляция готовых элементов управления AciveX

Деинсталляцию элемента ActiveX следует начинать с удаления ссылки на него. Для этого надо открыть искомый файл DPK, убрать ненужные элементы и перекомпилировать пакет. После этого элемент (или элементы) исчезнут со страницы Палитры компонентов. Если вы уверены в том, что этот элемент больше никем не используется, можете удалить информацию о нем из системного реестра (кнопка **Remove** диалогового окна **Import ActiveX**). Только после этого можно удалить с диска сам файл OCX или динамическую библиотеку, в которой он содержится.

3.3.3. Пример инсталляции элемента управления *TWebBrowser*

Рассмотрим в качестве примера использование элемента `TWebBrowser` из состава Microsoft Internet Controls (файл SHDOCVW.DLL). Он представляет собой ядро браузера от Microsoft и с большой долей вероятности уже есть

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 ранее операции, в вашем распоряжении будет файл SHDOCVW_TLB.PAS. В нем среди прочего содержится описание интерфейса `IWebBrowser`. Помимо интерфейса, в этом же файле есть 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 добиться от них с точки зрения Delphi сходного поведения. 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 — обращаясь к методам соответствующего класса Delphi.

Методы интерфейса `IWebBrowser` обеспечивают выполнение основных операций для Internet-браузера. Интерфейс `IWebBrowser` описан в модуле SCHDocVv.pas, методы интерфейса представлены в листинге 3.1.

Листинг 3.1. 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;
```

Не правда ли, все названия свойств и методов `IWebBrowser` просты и понятны? Теперь применим полученные знания 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 (рис. 3.4). 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`.

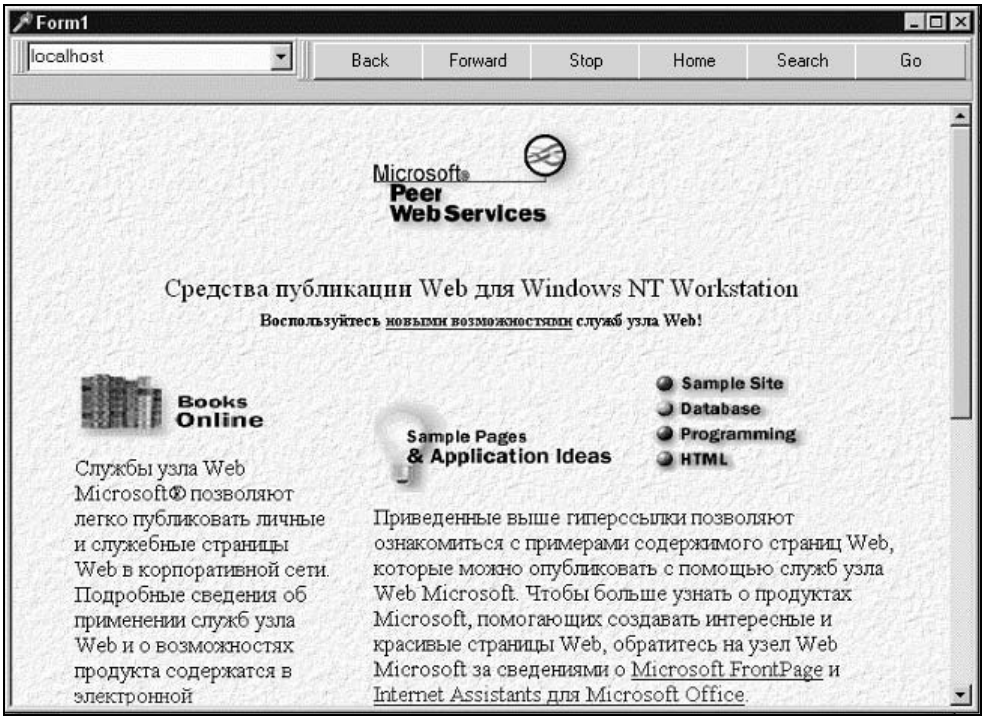


Рис. 3.4. Браузер, собранный на основе элемента управления ActiveX фирмы Microsoft

Код методов-обработчиков приведен в листинге 3.2.

Листинг 3.2. 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;
```

После компиляции и запуска приложения мы получим работоспособный Internet-браузер с набором важнейших функций (см. рис. 3.4). При этом вся наша работа — это связывание методов компонента ActiveX с кнопками.

Приложив к этому примеру еще немного усилий, вы получите полноценный браузер. Он может пригодиться вам для отладки приложений, примеры которых рассматриваются далее в этой книге.

3.4. Разработка собственных компонентов ActiveX

Технологии на основе COM настолько же привлекательны, насколько и сложны. Точнее сказать — не сложны, а трудоемки. Для того чтобы самому научиться строить объекты, нужно потратить немало времени. Поэтому и опытные пользователи, и новички делают это в Delphi при помощи целого набора мастеров. Сразу оговоримся, что мастер для создания компонентов ActiveX с нуля не предусмотрен. Вместо этого вы можете выбрать один из двух вариантов — превращение в элемент управления одного из обычных компонентов Delphi или целой формы.

3.4.1. Превращение компонентов Delphi в компоненты ActiveX

Первая из предоставляемых Delphi возможностей — превращение с помощью мастера любого оконного компонента (потомка класса `TWinControl`) в компонент ActiveX. Этот мастер становится доступным при выборе значка **ActiveX Control** на странице **ActiveX** Репозитория Delphi (рис. 3.5), но не ждите от него слишком многого — он умеет только то, что умеет.

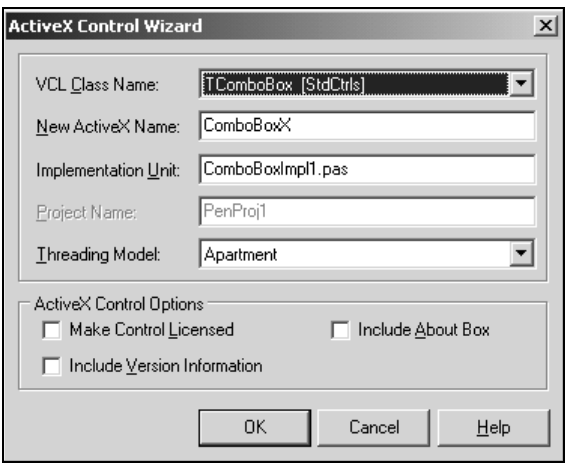


Рис. 3.5. Мастер преобразования компонентов в ActiveX

Использование мастера предельно просто.

1. Вы выбираете компонент VCL из списка **VCL Class Name**.
2. Мастер предлагает вам имена для будущего компонента ActiveX и связанных с ним файлов. Вы вольны принять это имя или задать собственный вариант в полях **New ActiveX Name**, **Implementation Unit**, **Project Name**.
3. Из списка **Threading Model** выбираете способ взаимодействия объекта и клиента *(более подробную информацию об этом см. в гл. 1)*.
4. Можно установить три дополнительных флажка: **Include About Box** — включение в проект диалогового окна с информацией о разработчике; **Include Version Information** — включение информации о версии элемента (этого требуют некоторые контейнеры); наконец, если вы не хотите свободного распространения своего продукта, вы можете включить в него лицензионную информацию, и пользоваться им в режиме разработки (т. е. во всех средах вроде Delphi или Visual Basic) смогут только пользователи, имеющие ключ. Для этого отметьте флажок **Make Control Licensed**.

После того как вы отредактируете предложенные параметры, мастер автоматически сгенерирует все необходимые файлы:

- собственно проект (напомним, что элементы ActiveX всегда оформляются в виде библиотек DLL — в данном случае с расширением OCX);
- библиотеку типов вместе с ее представлением на Object Pascal;
- и еще один файл с исходным текстом — файл реализации.

Файл реализации и описанные в нем классы тоже играют роль мостика, но если в случае TLB класс `TOleControl` обеспечивал связь между возможностями ActiveX и требованиями среды Delphi, то здесь потомки класса `TActiveXControl` устанавливают соответствие между бывшим компонентом и его новыми "хозяевами" — контейнерами, в которых будет размещаться созданный ActiveX.

Мы заостряем внимание на этом, потому что в файлах вы найдете два одноименных класса (скажем, при работе с компонентом `TCheckBox` это будет класс `TCheckBoxX`). Но тот из них, который является потомком `TOleControl`, нужен при импорте ActiveX, а потомок `TActiveXControl` — при экспорте.

Итак, после сохранения всех созданных файлов, компонент ActiveX готов к экспорту и использованию в других приложениях.

3.4.2. Преобразование форм в формы ActiveX

Более интересной представляется возможность создать в Delphi комплексный элемент управления, состоящий из многих простых. Она служит предметом зависти специалистов, работающих в других средах. Можно сказать, вы мало кого осчастливите, преобразовав в ActiveX стандартный компонент Delphi. Если вы разработали свой, интереса будет больше; но почти все нужные идеи уже реализованы. С другой стороны, перебирая десятки уже созданных элементов управления, вы почти в каждом найдете какой-то маленький штрих или деталь, из-за которых он не подходит именно вам. И только создание ActiveX на базе формы позволит сочетать богатство возможностей с простотой реализации. Эта технология называется *ActiveForms*.

Попытаемся опять совместить приятное с полезным и подберем для иллюстрации ActiveForms практический пример. Многим, кто создает графические приложения, нужны возможности выбора пользователем параметров пера и кисти. В то же время, переносить рутинный код из приложения в приложение не хочется. Выход — объединить его внутри нашего элемента управления.

Итак, для его создания необходимо выполнить описанную дальше последовательность действий.

Создается форма ActiveX в Delphi с помощью специализированного мастера. Для его вызова в Репозитории Delphi на странице **ActiveX** выберите значок **ActiveForm** (рис. 3.6).

В поле **New ActiveX Name** введите имя класса новой формы или примите предложенное по умолчанию. Далее согласитесь с предлагаемыми по умолчанию или измените имена модуля формы и проекта. В завершение выберите модель взаимодействия объекта и клиента в списке **Threading Model** *(более подробную информацию об этом см. в гл. 1)*.

Мы изменим имя нового элемента управления — PenX. Обратите внимание на то, что синхронно видоизменяются и остальные имена.

После этого сохраните проект. Созданный проект состоит из файла PENXIMPL.PAS (код реализации нашего компонента ActiveX), связанного с ним файла библиотеки типов PENXPROJ_TLB.PAS, а также файла проекта PENXPROJ.DPR (в нем содержится заголовок DLL).

Теперь для творчества нам доступна вся поверхность формы TPenX (модуль PENXIMPL.PAS).

Поместите на нее два компонента TComboBox и один — TTrackBar. Расположите их, как вам удобно (но примерно так, как показано на рис. 3.7), и уменьшите размеры формы, чтобы наш элемент ActiveX не получился слишком большим.

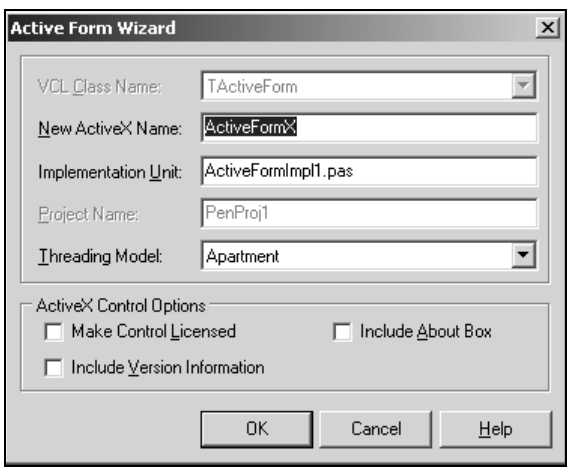


Рис. 3.6. Диалоговое окно **ActiveForm Wizard**

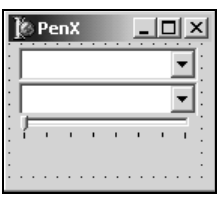


Рис. 3.7. Внешний вид объекта `TPenX`

Раскрывающиеся списки предназначены для выбора цвета и стиля пера, а компонент `TTrackBar` — для выбора его толщины. Свойства `Min` и `Max` компонента `TUpDown` установите в 1 и 8 соответственно. Списки будут отображать фиксированные свойства пера, поэтому придадим их свойствам

Style значения csOwnerDrawFixed и опишем для них обработчики событий (листинг 3.3).

Листинг 3.3. Методы-обработчики стандартных компонентов, размещенных на форме ActiveX

```
const DefColors : array [0..15] of TColor =(clBlack,
  clMaroon, clGreen, clOlive, clNavy, clPurple, clTeal, clGray, clSilver,
  clRed, clLime, clYellow, clBlue, clFuchsia, clAqua,
  clWhite);

var GblWidth: Integer;

procedure TPenX.FormCreate(Sender: TObject);
var i: Integer;
begin
 for i := Low(DefColors) to High(DefColors) do
  ClrComboBox.Items.Add(IntToStr(i));
 ClrComboBox.ItemIndex := 0;

 for i := 0 to 7 do
  StyleComboBox.Items.Add(IntToStr(i));
 StyleComboBox.ItemIndex := 0;
end;

procedure TPenX.ClrComboBoxDrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
 with ClrComboBox.Canvas do
 begin
  Brush.Color := DefColors[Index];
  FillRect(Rect);
 end;
end;

procedure TPenX.StyleComboBoxDrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var FPos : Integer;
begin
 FPos := Rect.Top + (Rect.Bottom – Rect.Top) div 2;
```

```
 with StyleComboBox.Canvas do
 begin
  Brush.Color := clWhite;
  FillRect(Rect);
  Pen.Style := TPenStyle(Index);
  Pen.Width := WidthTrackBar.Position;
  MoveTo(Rect.Left, FPos);
  LineTo(Rect.Right, FPos);
 end;
end;

procedure TPenX.ComboBoxChange(Sender: TObject);
begin
 (Sender as TWinControl).Repaint;
 if FEvents<>nil then FEvents.OnPenChanged;
end;

procedure TPenX.WidthTrackBarChange(Sender: TObject);
begin
 if WidthTrackBar.Position>1 then StyleComboBox.ItemIndex:=0;
 ComboBoxChange(StyleComboBox);
end;
```

При нажатии на кнопку списка `ClrComboBox` в нем должен появляться перечень 16 основных цветов; при нажатии на кнопку списка `StyleComboBox` — список возможных типов линий.

Формально элемент ActiveX уже готов. Однако для получения от него какой-то пользы недостает интерфейса с пользователем. Ведь нам выбор цветов и стилей нужен не сам по себе, а для передачи их пользователю.

В меню **View** среды разработки выбираем пункт **Type Library**. В появившемся окне Редактора библиотеки типов открываем интерфейс `IpenX`.

Примечание

Обратите внимание на автоматически созданные методы интерфейса `IPenX`. Все они обеспечивают выполнение стандартных функций элемента управления ActiveX и базируются на соответствующих методах интерфейсов, описанных в табл. 3.1.

Добавим к интерфейсу три свойства — `PenWidth`, `PenStyle` и `PenColor`. *Более подробную информацию о библиотеке типов и ее Редакторе см. в гл. 1.*

Новые свойства отобразятся как три пары методов — для чтения и для записи (рис. 3.8).

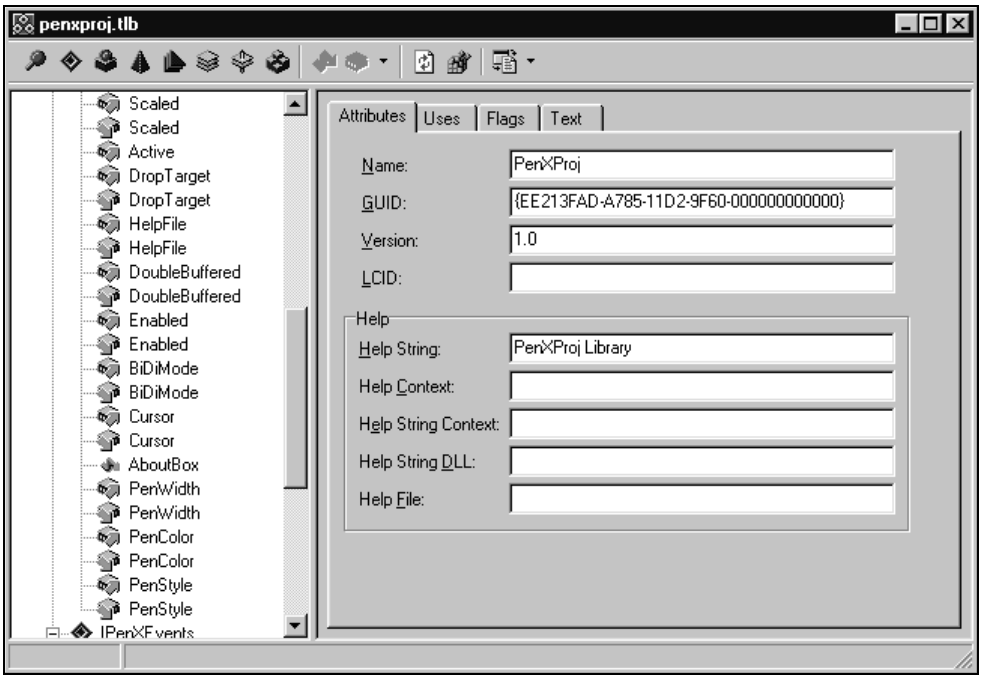


Рис. 3.8. Редактор Библиотеки типов с добавленными в интерфейс `IPenX` свойствами

Нажмите кнопку **Refresh Implementation** на верхней панели инструментов — при этом в обоих файлах проекта появятся описания свойств, а в PENXIMPL.PAS — заготовки кода для программной реализации свойств.

Примечание

В Delphi предусмотрен и другой путь добавления свойств в ActiveX — с помощью меню **Edit | Add to Interface**.

Приведем реализацию только для методов чтения (листинг 3.4). Программная переустановка значений этих свойств не планируется.

Листинг 3.4. Методы чтения новых свойств интерфейса `IPenX`

```
function TPenX.Get_PenColor: Integer;
begin
 Result := DefColors[ClrComboBox.ItemIndex];
```

```
end;

function TPenX.Get_PenStyle: Integer;
begin
 Result := StyleComboBox.ItemIndex;
end;

function TPenX.Get_PenWidth: Integer;
begin
 Result := WidthTrackBar.Position;
end;
```

Мы стали еще на один шаг ближе к цели — можно приостановить дальнейшее развитие элемента, скомпилировать его, установить, увидеть в Инспекторе объектов три созданные нами свойства и дальше обращаться к ним. Но когда пользователь должен переустанавливать параметры своего пера? Он пока ничего не знает о событиях, происходящих внутри элемента.

Это и будет последним штрихом.

Обратимся снова к просмотру библиотеки типов, на сей раз — к интерфейсу `IPenXEvents`. К нему нужно добавить не свойство, а метод, и назвать его `OnPenChanged`. В отличие от свойств, в данном случае появляется только ссылка на метод, но не заготовка для него. Реализовывать реакцию на событие `OnPenChanged` будем не мы, а программист, использующий наш элемент ActiveX, поэтому наша задача — вовремя инициировать наступление этого события. Такой код нужно поставить везде, где происходит изменение хотя бы одного из параметров пера:

```
procedure TPenX.ComboBoxChange(Sender: TObject);
begin
 if FEvents <> nil then FEvents.OnPenChanged;
end;
```

Элемент управления `ActiveForm` в целом готов (рис. 3.9).

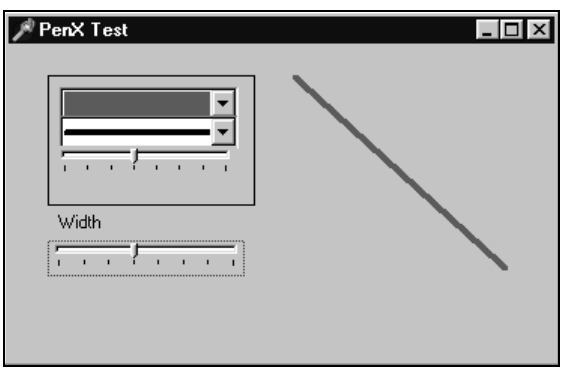


Рис. 3.9. Внешний вид элемента управления `PenX` в тестовом приложении

Осталось зарегистрировать элемент, включить его в состав одного из пакетов, и... наслаждаться возможностями, которые предоставляет вам компонентный подход к программированию!

Резюме

Разработка элементов управления ActiveX базируется на основе использования возможностей 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 использования 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, Delphi.

При помощи классов и мастеров Delphi вы можете создавать собственные элементы управления или формы и распространять их.

Глава 4

Технология COM+ (Microsoft Transaction Server)

Для распределенных приложений большое значение имеют вопросы обеспечения надежности, производительности, масштабируемости. Программный продукт Microsoft Transaction Server (MTS) предназначен для поддержки систем обработки транзакций и предоставляет среду для создания высокопроизводительных и надежных распределенных приложений для Internet и intranet.

MTS может устанавливаться и работать на компьютерах с операционными системами Windows 95/98, Windows NT, Windows 2000. Для семейства Windows 2000 продукт был переработан и получил название COM+.

Технология MTS базируется на возможностях COM и обеспечивает поддержку распределенных приложений на компонентной основе. Объекты транзакций MTS обладают основными свойствами объектов COM. Кроме этого, объекты транзакций реализуют специфические возможности, присущие только объектам MTS:

- управление транзакциями;
- безопасность;
- пулинг ресурсов;
- пулинг объектов.

Примечание

Система безопасности транзакций MTS не реализована для операционных систем Windows 95/98 вследствие объективных ограничений, накладываемых этими системами.

Для управления объектами транзакций и настройки параметров MTS используется ряд приложений и утилит.

- Координатор распределенных транзакций (Distributed Transaction Coordinator — DTC) представляет собой службу, управляющую транзак-

циями на низком уровне с использованием протокола двухфазной фиксации транзакций.

- Административное приложение MTS Explorer позволяет настраивать параметры среды MTS, хранимые в системном реестре; управлять пакетами и ролями MTS.
- Утилиты MTS предназначены для работы в командной строке или batch-файле.
- Исполняемый файл MTX.EXE реализует автоматические транзакции, обеспечивает их безопасность и активизацию (Just-In-Time — JIT).

Помимо перечисленных средств, для управления MTS можно использовать и стандартные системные средства. При этом для выполнения любых операций необходимо иметь полномочия администратора на данном компьютере.

К сожалению, обсуждение вопросов установки компонентов и настройки среды MTS для сервера и клиентов выходит за рамки этой книги.

В этой главе рассматриваются вопросы использования технологии MTS в Delphi. Перечислим основные темы:

- функционирование объектов транзакций;
- управление объектами транзакций;
- управление транзакциями;
- безопасность транзакций;
- управление ресурсами.

4.1. Как работает MTS

Итак, Microsoft Transaction Server — это совокупность программных средств, обеспечивающая разработку, распространение и функционирование распределенных приложений для Internet и intranet. В состав MTS входят следующие программные продукты:

- программное обеспечение промежуточного уровня, управляющее функционированием объектов транзакций во время выполнения;
- утилита MTS Explorer, позволяющая управлять объектами транзакций;
- интерфейсы прикладного программирования;
- средства управления ресурсами.

Стандартная программная модель приложений, использующих MTS, представляет собой трехзвенную архитектуру распределенных приложений, состоящую из серверов, клиентов и ПО промежуточного уровня. При этом бизнес-логика приложения сконцентрирована в *объектах транзакций MTS*,

а ПО промежуточного уровня, управляющее этими объектами, построено с использованием компонентной модели.

Разработчики, использующие MTS в своих приложениях, создают объекты бизнес-логики, удовлетворяющие требованиям, предъявляемым к объектам MTS; затем компилируют и устанавливают их в среде MTS при помощи пакетов.

Пакет MTS представляет собой контейнер, обеспечивающий группировку объектов в целях защиты данных, улучшения управления ресурсами и увеличения производительности. Управление пакетами MTS осуществляется утилитой MTS Explorer.

4.1.1. Объект MTS

Поскольку технология MTS основана на COM, объекты MTS должны удовлетворять основным требованиям, предъявляемым к объектам COM. Помимо этого, объекты MTS имеют ряд особенностей:

- объект должен быть реализован в составе внутреннего сервера (динамическая библиотека);
- объект должен содержать ссылку на библиотеку типов MTS;
- объект должен использовать только стандартный механизм маршаллинга COM;
- объект должен включать в себя интерфейс `IobjectControl`.

В соответствии с общими принципами работы MTS создаваемый объект может быть двух типов:

- stateful (с разделяемым состоянием);
- stateless (с неразделяемым состоянием).

Выбор типа объекта зависит от конкретной задачи и его назначения.

Подобно любому объекту COM, объекты MTS могут сохранять внутреннее состояние при многократном применении одного экземпляра. При каждом использовании объект сохраняет текущие значения своих свойств. При последующем вызове объект может вернуть запрашивающему клиенту текущее состояние. Такой объект имеет тип *stateful*. Объекты этого типа позволяют решать более широкий спектр задач.

Сохранение состояния объекта требует от него постоянной активности и сохранения таких ценных ресурсов, как, например, соединение с базой данных. На практике это означает не что иное, как работу с глобальными переменными, потому что именно в них хранится промежуточное состояние объекта.

Если объект не может сохранять свое промежуточное состояние, то это объект типа *stateless*. Объекты этого типа более эффективны.

Когда транзакция успешно завершена или прервана, все объекты, вовлеченные в транзакцию, деактивируются и тем самым теряют информацию о своем состоянии, приобретенную во время транзакции. Это помогает убедиться в изоляции транзакции и согласованности базы данных, а также освобождает ресурсы сервера для использования другими транзакциями. Завершение транзакции позволяет MTS деактивировать объект и обновить ресурсы.

4.1.2. Транзакции

Возможность объекта MTS "жить" внутри своей собственной транзакции или быть частью большой группы аналогичных объектов, которые принадлежат одной транзакции, является огромным преимуществом MTS. Это позволяет применять компонент для решения различных задач таким образом, что разработчики могут использовать код заново без модернизации логики приложения.

Транзакции MTS гарантируют, что:

- все изменения в одной транзакции будут приняты, либо она будет возвращена в свое предыдущее состояние;
- транзакция правильно и однозначно преобразует состояние системы;
- одновременные транзакции не видят частичные и несохраненные изменения, которые могут создавать конфликты;
- подтверждение изменений управляемых ресурсов (таких как записи баз данных) защищает от ошибок, включая ошибки сети и процессов;
- регистрация транзакций позволяет восстанавливать исходное состояние системы даже после ошибок на дисках.

Клиенты могут использовать прямой контроль над транзакциями с помощью контекста объекта, используя интерфейс `ITransactionContext`. Однако MTS может и сам для простоты работы клиента управлять транзакциями автоматически.

Применяя транзакции MTS, вы можете использовать бизнес-логику приложения в серверных объектах. Серверные объекты способны реализовывать бизнес-логику таким образом, что клиенту не надо будет ничего знать о ее правилах.

Атрибуты транзакций можно устанавливать тремя способами:

- на этапе разработки;
- с помощью редактора библиотеки типов;
- в среде MTS Explorer.

Завершение транзакции по умолчанию происходит спустя промежуток времени, указанный в параметре `transaction timeout`, который задается для

каждого объекта отдельно с помощью утилиты MTS Explorer. По умолчанию это время равно 60 секундам. По истечении этого времени, если транзакция не успела завершиться, она автоматически прерывается.

4.1.3. Контекст объекта MTS

Для каждого объекта транзакции MTS сервер транзакций автоматически создает специальный объект, который носит название *объекта контекста транзакции* или *контекста объекта MTS*. Выполнение функций контекста обеспечивается интерфейсом `IObjectContext`.

Два метода интерфейса определяют способ выхода объекта из транзакции.

Метод `SetComplete` сообщает транзакции, что он готов к завершению своей работы в транзакции.

Использование метода `SetAbort` означает, что исполнение кода объекта привело к возникновению обстоятельств, препятствующих успешному завершению транзакции.

После использования любого из этих двух методов объект завершает свое участие в транзакции (рис. 4.1).

Методы `EnableCommit` и `DisableCommit` сообщают о текущем состоянии объекта. Метод `EnableCommit` информирует о том, что объект позволяет завершить транзакцию несмотря на то, что он продолжает работать.

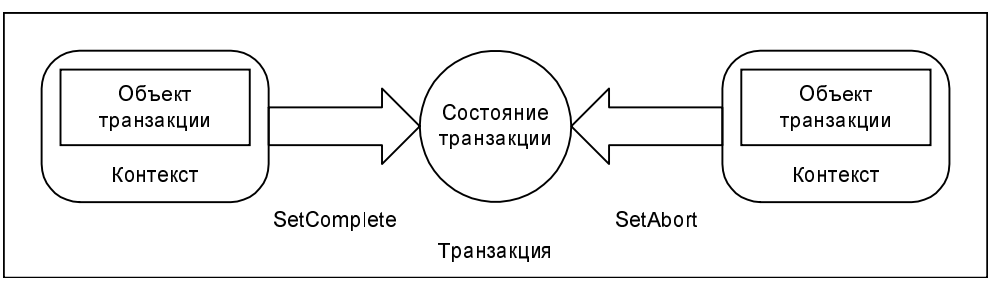


Рис. 4.1. Роль контекста объекта MTS

Вызов метода `DisableCommit` показывает, что в настоящий момент текущее состояние объекта не позволяет завершить транзакцию. При попытке завершить транзакцию после вызова этого метода, транзакция будет прервана.

При помощи перечисленных методов объект контекста обеспечивает среду MTS информацией о состоянии объекта транзакции.

Например, распределитель ресурсов может использовать контекст объекта MTS для обеспечения сервисов на основе транзакций. Пусть объект выполняется внутри транзакции, которая зарезервировала соединение с базой данных, используя провайдер ADO. Это соединение автоматически органи-

зует транзакцию. Все изменения в базе данных, использующие это соединение, становятся частью транзакции и затем либо принимаются, либо отвергаются.

Дополнительно разработчики могут применять еще несколько методов интерфейса `IObjectContext`.

При использовании объекта MTS в открытой транзакции функция `IsInTransaction` объекта контекста возвращает `True`.

Если для объекта MTS или пакета MTS, содержащего этот объект, включен режим защиты данных, то метод `IsSecurityEnabled` возвращает значение `True`.

Если приложение, использующее объект MTS, исполняет некоторую роль MTS *(см. разд. 4.1.4)*, то метод `IsCallerInRole` возвращает значение `True`.

4.1.4. Безопасность данных в MTS

Одним из удобных сервисов, предоставляемых MTS, является обеспечение доступа к компонентам (и даже к отдельным интерфейсам) в зависимости от прав, которыми обладает клиент. Безопасность MTS включает в себя:

- декларативную защиту данных;
- программную защиту данных.

В обоих случаях для обеспечения безопасности среда MTS использует роли MTS. *Роль MTS* — это абстрактное представление некоторой совокупности пользователей. В него могут входить отдельные пользователи или группы пользователей. При помощи приложения MTS Explorer администратор создает необходимые роли и заносит в них пользователей и группы. Каждая роль наделяется необходимыми правами. При этом для проверки принадлежности к группе можно использовать механизм аутентификации Windows.

Декларативная защита данных

Декларативная защита данных создается на этапе настройки среды MTS и выполняется средствами утилиты MTS Explorer. Она заключается в ограничении доступа к тому или иному объекту или пакету для пользователей и групп, являющихся исполнителями тех или иных ролей.

По умолчанию в среде MTS настроен пакет System Package, для которого предопределены две роли: *администратора (Administrator)* и *читателя (Reader)*. Перед началом работы необходимо связать роль администратора с хотя бы одной учетной записью.

Примечание

Для операционных систем Windows 95/98 настройка безопасности невозможна вследствие ограниченной функциональности этих операционных систем.

Программная защита данных

Программная защита данных обеспечивается объектом контекста и методами `IsSecurityEnabled` и `IsCallerInRole` интерфейса `IObjectContext` этого объекта. Программная защита данных проектируется на этапе разработки приложения и исполняется во время работы приложения, использующего данный объект MTS.

Когда приложение пытается активизировать некоторый объект MTS, то вызывается метод `IsCallerInRole`. В качестве параметра метода передается роль, исполняемая приложением. Если объект или пакет MTS играет эту роль, его использование разрешается, и метод возвращает значение `True`.

Если же несколько объектов MTS используются в рамках одного процесса, метод `IsCallerInRole` возвращает `True` всегда. В этом случае для более точной идентификации применяется метод `IsSecurityEnabled`.

Примечание

Для приложений, которые требуют более строгой безопасности, контекст объекта применяет интерфейс `ISecurityProperty`, чьи методы позволяют возвращать идентификатор Windows security identifier (SID) для прямого вызова и создания объекта.

4.1.5. Ресурсы

Для управления ресурсами MTS использует три способа:

- активизацию Just-in-time;
- пулинг ресурсов (Resource pooling);
- пулинг объектов (Object pooling).

Рассмотрим конкретнее каждый из способов.

Активизация Just-in-time

Способность объекта быть деактивированным и повторно активированным, пока клиент сохраняет ссылку на него, называется *активизацией just-in-time*.

В процессе работы приложения часто бывает необходимо использовать один экземпляр объекта MTS несколько раз через определенные промежутки времени. При обращении к объекту он активизируется, а некоторое время после прекращения использования приложение удерживает ссылку на этот объект.

Когда создается объект COM как часть среды MTS, также формируется соответствующий контекст объекта. Этот контекст объекта существует в течение всего времени жизни соответствующего объекта MTS, на протяжении

одного или нескольких циклов. MTS использует контекст объекта для сохранения информации об объекте при его деактивизации.

Объект COM создается в неактивном состоянии и становится активным только после запроса клиента.

При переходе объекта в неактивное состояние MTS уничтожает все его ресурсы и в том числе, например, соединение с базой данных.

Объект MTS становится неактивным при возникновении следующих событий.

- Вызов методов `SetComplete` или `SetAbort` интерфейса `IObjectContext`. Объект вызывает метод `SetComplete`, когда он успешно завершил свою работу и нет необходимости сохранять внутреннее состояние объекта для следующего вызова клиента. Объект вызывает `SetAbort`, если нет возможности успешно завершить работу и отсутствует необходимость сохранения состояния объекта. Затем объект возвращается в состояние, предшествующее этой транзакции. При нормальной реализации stateless-объекта деактивизация происходит после вызова каждого метода.
- Транзакция сохраняется либо прерывается. После этого могут продолжить свое существование только те объекты, которые имеют ссылку на клиентов за пределами этой транзакции. Последующий вызов этих объектов повторно активирует их и служит причиной выполнения методов объекта в следующей транзакции.
- Последний клиент освобождает объект. При этом контекст объекта тоже освобождается.

Пулинг ресурсов

После освобождения ресурсов при деактивизации объекта MTS, они становятся доступными для других серверных объектов. Этот процесс называется *пулингом ресурсов*.

Рассмотрим, например, соединение с базой данных через провайдер ADO. Известно, что выделение ресурсов, открытие и закрытие соединения с базой данных занимает довольно много времени. Частое повторение этой операции различными объектами MTS применительно к одной базе данных вызовет повышенный расход ресурсов.

Именно в таких случаях и используется пулинг ресурсов. Соединение с базой данных, больше не используемое одним серверным объектом, может быть использовано другим объектом.

Для выполнения задач пулинга ресурсов MTS использует *распределитель ресурсов*.

Освобождение ресурсов

Обычно освобождение ресурсов объекта делается при помощи вызова методов `SetComplete` и `SetAbort` после обслуживания запроса клиента. Эти методы освобождают ресурсы, зарезервированные распределителем ресурсов MTS.

В то же самое время, необходимо освобождать ссылки на другие ресурсы, включая ссылки на другие объекты (и объекты MTS, и контексты объектов) и память, занятую экземплярами компонентов. Этого не следует делать лишь тогда, когда надо сохранить информацию о состоянии между вызовами клиентов.

Пулинг объектов

MTS реализует не только пулинг ресурсов, но и *пулинг объектов*. Правда, эта возможность доступна только в рамках технологии COM+, работающей под управлением операционной системы Windows 2000. В MTS более старых версий пулинг объектов лишь декларирован для будущего использования.

Суть этого механизма проста.
При выполнении приложений в среде MTS создается специальный пул объектов (рис. 4.2).

Для управления пулом и размещения в нем объектов используется интерфейс `IObjectControl`. Если объект предназначен для использования в пулинге, то метод `CanBePooled` интерфейса должен возвращать значение `True`.

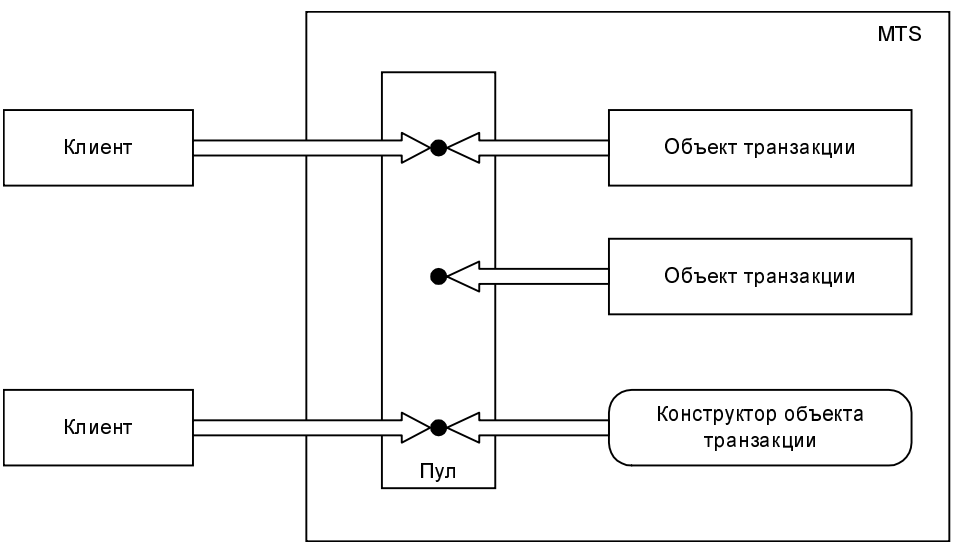


Рис. 4.2. Схема пулинга объектов MTS

После деактивизации такого объекта сервер MTS помещает его в пул. Объекты внутри пула доступны для немедленного использования любыми другими запросами клиентов. В случае если объект запрошен, но пул объектов пуст, MTS автоматически создает новый экземпляр объекта.

4.2. Создание приложений MTS в Delphi

Delphi предоставляет разработчикам широкий спектр средств для создания распределенных приложений, применяющих технологии MTS.

Это в первую очередь объекты транзакций MTS, для создания которых предназначен специализированный мастер.

Для разработки серверных частей приложений можно использовать модули данных MTS, обеспечивающие обработку вызовов клиентов в среде MTS.

В Delphi также существуют средства управления ресурсами, которые можно применять при написании программ.

Рассмотрим перечисленные инструменты более детально.

4.2.1. Объекты транзакций MTS

Объекты транзакций MTS представляют собой объекты COM *(см. гл. 1)*, обладающие рядом специфических возможностей. Эти возможности реализованы благодаря использованию интерфейсов MTS. Основными среди них являются интерфейсы `IObjectControl` и `IObjectContext`.

Объекты транзакций MTS обычно применяются для реализации небольших блоков бизнес-логики приложения. При этом один объект может работать с одной транзакцией монопольно или использовать ее совместно с другими объектами. Это зависит лишь от выбранного способа реализации бизнес-логики и ее алгоритмов.

Класс *TMTSAutoObject*

Возможности объекта транзакции в Delphi реализованы в базовом классе `TMTSAutoObject` (рис. 4.3).

Его свойства и методы обеспечивают выполнение основных функций объекта MTS:

- уведомление транзакции о состоянии объекта;
- обеспечение программной защиты данных;
- пулинг объектов (только для технологии COM+).

Все эти возможности уже рассмотрены нами, и так как методы класса `TMTSAutoObject` полностью совпадают с методами уже известных нам интерфейсов, остановимся лишь кратко на их описании.

Методы

```
procedure SetComplete;
procedure SetAbort;
```

вызывают одноименные методы интерфеса `IObjectContext` и обеспечивают сохранение или откат транзакции.

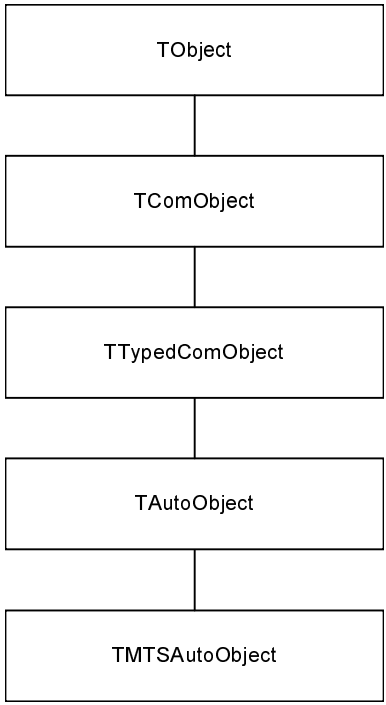


Рис. 4.3. Иерархия класса TMTSAutoObject

Методы

```
procedure DisableCommit;
procedure EnableCommit;
function IsInTransaction: Bool;
```

вызывают одноименные методы интерфейса `IObjectContext` и обеспечивают уведомление транзакции о состоянии объекта.

Методы

```
function IsCallerInRole(const Role: WideString): Bool;
function IsSecurityEnabled: Bool;
```

реализуют программную защиту данных. Параметр `Role` метода `IsCallerInRole` должен содержать имя запрашиваемой роли MTS.

Свойство

```
property Pooled: Boolean;
```

управляет доступностью объекта для пулинга. Если свойство имеет значение `True`, объект может помещаться в пул для многократного использования.

Создание объекта транзакции

Для создания нового объекта транзакции MTS в Delphi используется мастер **New Transactional Object**, который доступен в Репозитории на странице **ActiveX** (рис. 4.4).

Согласно требованиям, объекты транзакций MTS могут быть реализованы только в составе внутреннего сервера, поэтому при создании нового объекта транзакции с помощью мастера **New Transactional Object** тип проекта автоматически меняется на динамическую библиотеку ActiveX *(см. гл. 3)*.

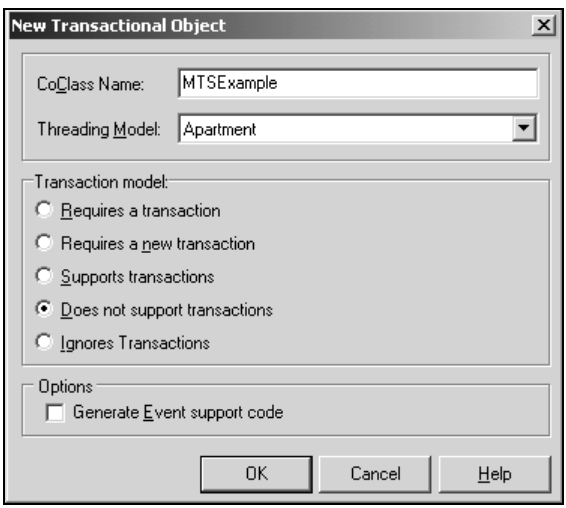


Рис. 4.4. Мастер создания объекта транзакции

В строке **CoClass Name** необходимо указать имя объекта транзакции, а точнее имя класса-оболочки (coclass), используемого для представления объекта в приложении.

Примечание

Классы `coclass` используются в Delphi для любых объектов COM или дочерних технологий. Более подробную информацию о роли классов-оболочек в приложениях, использующих COM, см. в *гл. 1*.

Список **Threading Model** позволяет выбрать способ взаимодействия сервера и запросов клиентов. Иначе это называется потоковой моделью. Возможные варианты модели детально обсуждались в *гл. 1*. Далее мы рассмотрим особенности применения различных моделей для объектов MTS.

Группа радиокнопок **Transaction Model** определяет способ поведения объекта в транзакции.

Флажок **Generate Event support code** разрешает или запрещает управление событиями объекта транзакции. Если флажок установлен, то при создании исходного кода класса объекта MTS автоматически будет создан интерфейс `IConnectionPointContainer`. Механизм уведомления о событиях (нотификации) подобен используемому для объектов Автоматизации. *Более детальная информация об этом содержится в гл. 2.*

После настройки параметров нового объекта и нажатия кнопки **OK** Delphi автоматически генерирует исходный код для класса нового объекта (листинг 4.1).

Листинг 4.1. Объявление класса объекта транзакции MTS

```
unit Unit1;
{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  ActiveX, Mtsobj, Mtx, ComObj, Project1_TLB, StdVcl;

type
  TMTSExample = class(TMtsAutoObject, IMTSExample)
  protected
    { Protected declarations }
  end;

implementation

uses ComServ;

initialization
  TAutoObjectFactory.Create(ComServer, TMTSExample, Class_MTSExample,
    ciMultiInstance, tmApartment);
end.
```

Как видите, представленный код вполне обычен для объектов COM и некоторые его варианты встречались уже не раз в предыдущих главах. Помимо собственно объявления класса, в секции `initialization` присутствует конструктор фабрики класса.

Одновременно с классом создается и интерфейс (в нашем примере это интерфейс `IMTSExample`), предназначенный для представления методов нового

класса. Этот интерфейс является дуальным (обратите внимание, на рис. 4.3 класс объекта MTS — потомок класса объекта Автоматизации).

Одновременно с объявлением класса создается и соответствующая библиотека типов. В ней, в соответствии с требованиями технологий COM и Автоматизации, описываются основной и диспинтерфейсы, а также `coclass` для нового класса.

Типы потоковой модели

Мастер создания объекта MTS требует указать тип потоковой модели. Рассмотрим их.

Модель `Single` требует выполнения всех создаваемых в ответ на запросы клиентов объектов в едином потоке. Эта модель принята по умолчанию для всех объектов COM.

Она обладает ограниченной масштабируемостью. Объекты, использующие эту модель и имеющие тип `stateful`, способны стать причиной тупиков и блокировок. Вы можете устранить эту проблему, применив объект типа `stateless` и вызвав метод `SetComplete` перед завершением запроса.

Модель `Apartment` создает для каждого нового экземпляра объекта отдельный поток, который используется на протяжении времени жизни объекта.

В рамках этой модели два объекта могут выполняться параллельно до тех пор, пока они находятся в различных действиях (activity).

Модель `Both` аналогична модели `Apartment`, за исключением того, что ответы объектов на запросы обрабатываются последовательно, а не параллельно. Эта модель рекомендуется в качестве основной при использовании пулинга объектов.

Примечание

Модель `Free`, используемая в других объектах COM, не применима к объектам, запускаемым в среде MTS, в силу поддержки в MTS так называемых действий (activities). *Более подробную информацию о действиях MTS см. в разд. 4.4.*

Поведение объектов MTS в транзакциях

Мастер создания объекта MTS требует указать тип модели транзакции. Всего доступны для использования пять типов.

- `Requires a transaction`. Объекты MTS обязаны выполняться внутри транзакции. Когда создается новый объект, его контекст наследует транзакцию от контекста клиента. Если у клиента нет транзакции, то MTS автоматически ее создает.
- `Requires a new transaction`. Объекты MTS обязаны выполняться внутри своих транзакций. Когда новый объект создается, MTS автоматически

порождает новую транзакцию для объекта, независимо от того, есть она или нет у клиента. Повторно объект никогда не запускается внутри своей собственной уже запущенной транзакции. Вместо этого система всегда создает независимую транзакцию для новых объектов.

- `Supports transactions`. Объекты MTS выполняются внутри их клиентской транзакции. Когда создается новый объект, его контекст наследует транзакцию от контекста клиента. Это позволяет нескольким объектам работать в пределах одной транзакции. Если у клиента нет транзакции, то новый контекст создаст ее.
- `Transactions Ignored`. При создании нового объекта связанный объект контекста транзакции создается без транзакции и независимо от ее существования. Применим только для технологии COM+.
- `Does not support transactions`. Зависит от используемой технологии. Для MTS действие аналогично `Transactions Ignored`. Для COM+ объект контекста транзакции создается без участия транзакции. Объект продолжает зависеть от транзакции, но не активизируется немедленно, если транзакция существует.

4.2.2. Удаленный модуль данных MTS

Удаленный модуль данных MTS позволяет создавать серверные части распределенных приложений. Он обеспечивает базовую функциональность сервера приложения. Созданный на его основе сервер приложения, будучи корректно зарегистрированным на компьютере сервера, выполняет следующие функции:

- служит контейнером объектов транзакций MTS;
- инкапсулирует бизнес-логику распределенного приложения, реализованную в наборе объектов транзакций MTS;
- обеспечивает обработку запросов клиентов в соответствии с заданной потоковой моделью;
- взаимодействует со средой MTS, обеспечивая совместно с ней работоспособность объектов MTS.

Базовая функциональность сервера приложения обеспечивается интерфейсом `IAppServer`, включенным в модуль данных.

На этапе разработки модуль данных MTS является контейнером для любых невизуальных компонентов.

Детальную информацию о работе с удаленными модулями данных и создании серверов приложений см. в главах ч. III.

Возможности удаленного модуля данных инкапсулированы в классе `TMTSDataModule`.

Для создания нового удаленного модуля данных MTS используется мастер **New Transactional Data Module Object**. Он вызывается при помощи Репозитория Delphi (рис. 4.5). Значок мастера **Transactional Data Module** располагается на странице **Multitier** Репозитория.

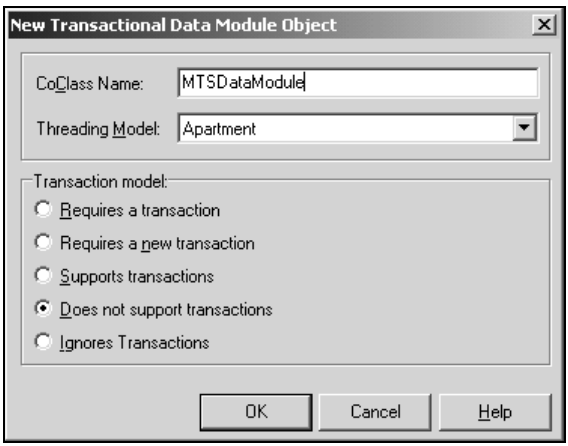


Рис. 4.5. Мастер создания удаленного модуля данных MTS

В строке **CoClass Name** необходимо указать имя модуля данных.

Список **Threading Model** позволяет выбрать способ взаимодействия сервера и запросов клиентов.

Группа радиокнопок **Transaction Model** определяет способ поведения объекта в транзакции.

После создания новый модуль данных доступен программисту. На этапе разработки он используется как обычный модуль данных. На нем можно размещать любые невизуальные компоненты и создавать исходный код для них.

Дополнительно к возможностям удаленного модуля данных, унаследованным от класса `TRemoteDataModule`, модуль данных MTS включает в себя методы интерфейса `IObjectContext`.

4.2.3. Распределители ресурсов

Delphi предоставляет для использования два распределителя ресурсов:

- *Распределитель ресурсов Broland Database Engine (BDE, процессор баз данных)*, (BDE resource dispenser). Управляет пулом соединений с базой данных для компонентов MTS, использующих для работы с базами данных возможности BDE. BDE представляет собой ПО промежуточного уровня, обеспечивающее доступ к нескольким типам баз данных.

- *Менеджер разделенных свойств (Shared Property Manager)* используется для разделения состояний среди нескольких объектов внутри одного серверного процесса.

При использовании менеджера разделенных свойств не надо добавлять дополнительный код в приложение. Он также устраняет конфликт имен с помощью *групп с разделяемыми свойствами (shared property groups)*, которые устанавливают уникальное пространство имен для содержащихся в них свойств.

Применяя менеджер разделенных свойств, вы прежде всего используете функцию `CreateSharedPropertyGroup` для создания группы с разделяемыми свойствами. Затем вы можете сохранить и восстановить все свойства этой группы. В дополнение, информация о состоянии может быть разделена среди всех объектов MTS, которые инсталлированы в том же пакете.

Следующий пример (листинг 4.2) показывает, как добавить код для поддержки менеджера разделенных свойств в объекте MTS.

Листинг 4.2. Пример использования менеджера разделенных свойств

```
unit Unit1;
{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  ActiveX, Mtsobj, Mtx, ComObj, Project1_TLB, StdVcl;

type
  TMTSExample = class(TMtsAutoObject, IMTSExample)
  private
    Group: ISharedPropertyGroup;
  protected
    procedure OnActivate; override;
    procedure OnDeactivate; override;
    procedure IncCounter;
  end;

implementation

uses ComServ;

procedure TMTSExample.OnActivate;
```

```
var Exists: WordBool;
    Counter: ISharedProperty;
begin
  Group := CreateSharedPropertyGroup('MyGroup');
  Counter := Group.CreateProperty('Counter', Exists);
end;

procedure TMTSExample.IncCounter;
var Counter: ISharedProperty;
begin
  Counter := Group.PropertyByName['Counter'];
  Counter.Value := Counter.Value + 1;
end;

procedure TMTSExample.OnDeactivate;
begin
  Group := nil;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TMTSExample, Class_MTSExample,
    ciMultiInstance, tmApartment);
end.
```

В этом примере в методе-обработчике `OnActivate` объекта `TMTSExample` создается группа с разделяемыми свойствами, которая названа `MyGroup`, а также разделяемое свойство `Counter` для этой группы.

Функция `CreateSharedPropertyGroup` используется для формирования менеджера группы с разделяемыми свойствами и самой группы свойств. Затем для создания свойства используется метод `CreateProperty` группы `MyGroup`.

Для получения значения свойства следует применять метод `PropertyByName` объекта группы.

Обратите внимание на то, что функциональность группы с разделяемыми свойствами определена в методах интерфейса `ISharedPropertyGroup`.

Примечание

Все объекты с разделяемым состоянием (stateful) должны быть запущены в одном и том же серверном процессе. Объекты с разделяемыми свойствами должны иметь одинаковые атрибуты активизации. Например, если один объект сконфигурирован для запуска в процессе клиента, а другой в процессе сервера, то эти объекты будут запущены в разных процессах, даже если они инсталлированы в одном пакете.

4.3. Тестирование и установка MTS компонентов

При настройке среды MTS среди многих параметров администратор может задавать время пребывания объекта транзакции в активном состоянии без вызовов со стороны клиентов. Это параметр `transaction timeout`. Его значение по умолчанию равно 60 секундам.

При отладке объектов MTS необходимо отключить этот параметр (присвоить значение, равное нулю), иначе объект может быть выгружен средой MTS, пока вы работаете с исходным кодом или значениями переменных в процессе отладки.

На стадии разработки MTS-компонента нельзя его перекомпилировать, пока он находится в памяти. В этом случае появится сообщение об ошибке "Cannot write to DLL while executable is loaded". Для устранения этой ситуации необходимо с использованием MTS Explorer установить в свойствах пакета параметр "shut down after being idle for 3 minutes", изменив соответственно время.

Для этого надо выполнить следующие действия.

1. В MTS Explorer щелкнуть правой кнопкой мыши по пакету, в котором инсталлирован интересующий вас объект транзакции MTS, и из всплывающего меню выбрать пункт **Свойства**.
2. В появившемся диалоговом окне выбрать вкладку **Advanced**.
3. Изменить время ожидания на 0.
4. Нажать кнопку **OK** для сохранения параметров и вернуться в среду MTS Explorer.

Для запуска объекта транзакции MTS в режиме отладки необходимо настроить параметры запуска (рис. 4.6).

1. В Delphi выбрать пункт меню **Run | Parameters**.
2. Указать в строке **Host Application** полный путь к файлу mtx.exe.
3. Напечатать в строке **Parameters**: ключ `p` и имя пакета, в котором установлен объект транзакции:

   ```
   /p:"TestPack"
   ```

   в этом примере TestPack — название пакета, в который инсталлирован объект.

После этого, расставив в нужных местах точки останова, можно запустить серверный объект и, посылая запросы от клиента, заниматься отладкой приложения.

Примечание

Каждый пакет запускается как единственный экземпляр исполняемого файла mtx.exe. При запуске нескольких пакетов это можно увидеть в Диспетчере задач, на странице **Процессы**.

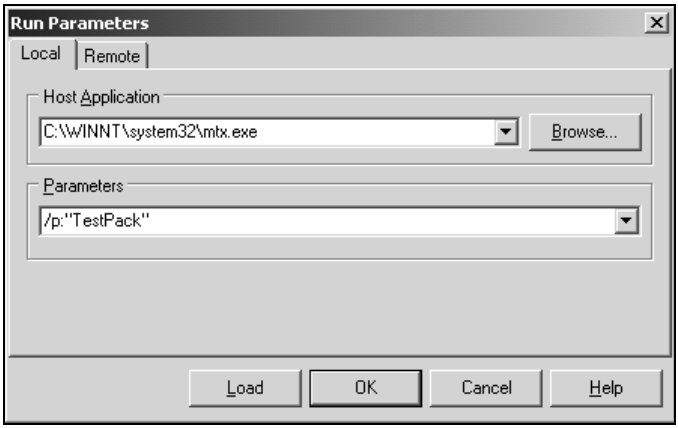


Рис. 4.6. Настройка параметров запуска объекта транзакции

Для установки из Delphi объекта транзакции в пакет необходимо выполнить следующие действия. Обратите внимание на то, что для Windows 2000 технология носит название COM+.

1. Выбрать пункт меню **Run | Install COM+ Objects**.
2. В появившемся диалоговом окне **Install COM+ Objects** (рис. 4.7) выбрать из списка объект MTS, который будет инсталлирован; для этого необходимо установить флажок слева от имени объекта.

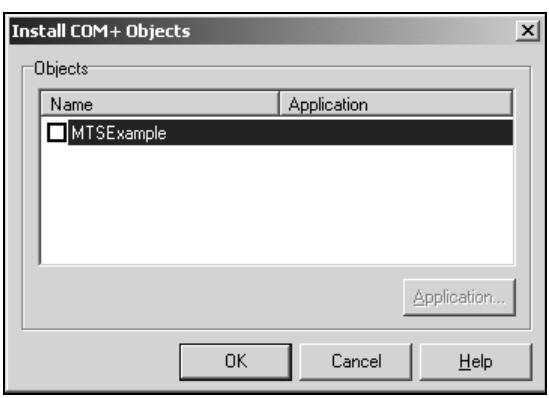


Рис. 4.7. Диалоговое окно выбора объекта транзакции MTS для инсталляции

3. При установке флажка автоматически появляется еще одно диалоговое окно **Install COM+ Objects** (рис. 4.8), в котором предлагается выбрать пакет, в который будет установлен объект транзакции MTS. Здесь необходимо выбрать **Install into new Application** для создания нового пакета или

Install into existing Application для инсталляции в уже существующий (выбрав его из списка).

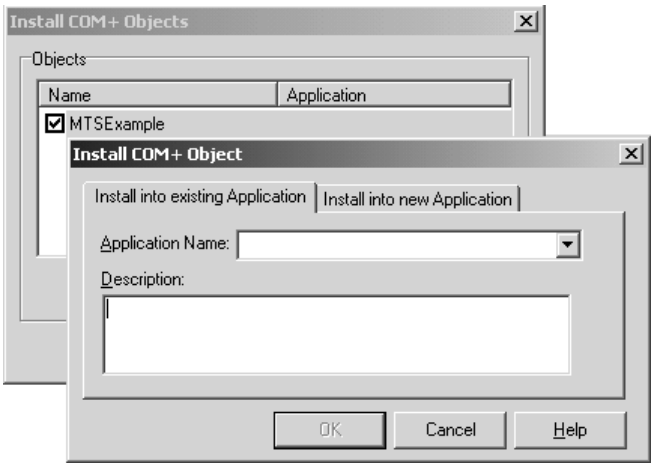


Рис. 4.8. Диалоговое окно выбора приложения для объекта транзакции MTS

4. Нажать кнопку **OK** для завершения операции.

Примечание

Один объект не может быть установлен в несколько пакетов.

4.4. Оптимизация работы с MTS

Разработать приложение, использующее MTS или COM+, формально просто. Более сложная задача — проектирование приложения, обладающего максимальной эффективностью благодаря использованию всех возможностей современной технологии. Для этого надо хорошо представлять механизм работы объектов COM и принципы работы с транзакциями.

4.4.1. Блокировка транзакций

Одним из основных принципов грамотного управления транзакциями является их изоляция. Для ее достижения используются различные механизмы блокировок влияния транзакций друг на друга. MTS реализует систему изоляции транзакций на высоком уровне. Пока данные в одной транзакции не будут обработаны, они не видны в другой транзакции. Это снижает производительность системы. Для решения возникшей проблемы необходимо свести время работы каждой транзакции к минимуму и не забывать

о правильном использовании методов, таких как `SetComplete` и `SetAbort` интерфейса `IobjectContext`.

4.4.2. Действия MTS

Для MTS одним из наиболее важных фундаментальных понятий является *действие MTS (activity)*. Очень часто на правильное использование действий не обращают серьезного внимания, хотя именно из-за неправильной работы с действиями возникают некоторые проблемы.

Действие MTS — это совместное функционирование совокупности объектов, в интересах единственного клиента. Действие может содержать объекты из разных пакетов. Каждый объект MTS существует только в одном действии, хотя действие может содержать несколько объектов. Каждая транзакция MTS существует только в одном действии, хотя действие может содержать несколько транзакций.

Программирование для MTS подразумевает, что объекты MTS не должны разделяться между действиями. Параллельное использование объектов внутри действия очень опасно, поскольку объект, работающий в интересах одного потока, может попытаться принять транзакцию, пока объект, работающий в интересах второго потока, находится в процессе выполнения работы внутри этой же транзакции. Если бы транзакция была действительно принята, то это привело бы к фиксации частично выполненной транзакции.

4.5. Пример простого объекта транзакции

После изучения основных принципов и понятий MTS создадим простой пример, в котором клиент будет запрашивать у сервера информацию о текущих проектах. В качестве источника данных используем базу данных Microsoft Access DBDEMOS.MDB. Это стандартная демонстрационная база данных, поставляемая совместно с Delphi. По умолчанию файл базы данных инсталлируется в папку ..\Program Files\Common Files\Borland Shared\Data.

В нашем простом примере сервер будет возвращать общее количество заказов, имеющихся в базе данных.

Создадим в Delphi новую группу проектов и первый проект используем для создания сервера.

4.5.1. Создание сервера

С помощью уже описанного мастера создадим модуль данных MTS, который назовем `DMTest`. При создании модуля данных в мастере **New Transactional Data Module Object** необходимо задать потоковую модель `Apartment` и модель транзакции `Supports transactions`.

Разместим здесь группу невизуальных компонентов, необходимых для организации доступа к базе данных через провайдер ADO Microsoft Jet 4.0 OLE DB Provider. В нашем простом случае это компонент соединения `TADOConnection` и компонент запроса `TADOQuery`. Настроим соединение и запрос. *Подробную информацию о работе с технологией ADO см. в гл. 7.*

Внимание!

При компиляции примера на вашем компьютере необходимо заново указать путь к базе данных, используя редактор свойства `ConnectionString` компонента `TADOConnection`. В вашем примере это свойство не заполнено, так как реальное местоположение файла базы данных может различаться.

Теперь в библиотеке типов удаленного модуля данных MTS для интерфейса `IDMTest` создадим метод `GetOrderCount` для обработки запроса клиента, а в модуле uDMTest представлена его реализация (листинг 4.3). Этот метод при помощи компонента запроса будет определять общее число заказов.

Листинг 4.3. Описание метода `GetOrderCount` в секции `implementation` модуля `uDMTest`

```
procedure TDMTest.GetOrderCount(out OrdCount: Integer);
begin
 try
  conDBDemos.Open;
  try
   quOrdCount.Open;
   OrdCount := quOrdCount.Fields[0].AsInteger;
   quOrdCount.Close;
  finally
   conDBDemos.Close;
  end;
  SetComplete;
 except
  OrdCount := DefOrdCount;
  SetAbort;
 end;
end;
```

Обратите внимание на использование методов `SetComplete` и `SetAbort`, которые реализованы и для класса `TMTSDataModule`.

Откомпилируем проект и затем установим его в новый пакет TestPack (см. рис. 4.7, 4.8).

4.5.2. Создание клиента

Для создания клиентского приложения добавим в группу проектов новое приложение. Назовем его `SimpleMTSClient`. Главная форма приложения представлена на рис. 4.9.

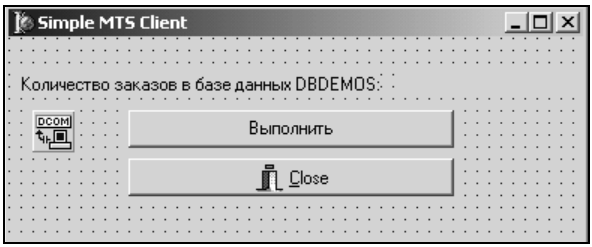


Рис. 4.9. Главная форма клиентского приложения MTS

Связь между сервером и клиентом можно организовать, применяя различные компоненты: `TDCOMConnection`, `TWebConnection`, `TSocketConnection`. Используем в примере компонент, работающий с Distributed COM, так как он наиболее прост в настройке и хорошо подходит для применения на одном компьтере с сервером или в локальной сети. Для его настройки достаточно указать в свойстве `ComputerName` имя компьютера сервера, а затем выбрать имя сервера из списка свойства `ServerName`.

Внимание!

При работе с этим примером необходимо самостоятельно заполнить значения свойств `ComputerName` и `ServerName` компонента `TDCOMConnection`.

Для отправки запроса на сервер используем метод-обработчик нажатия кнопки `buCalculate`. Исходный код крайне прост:

```
procedure TfmMain.buCalculateClick(Sender: TObject);
var OrdCount: Integer;
begin
 Screen.Cursor := crHourGlass;
 try
  conDCOM.Open;
  conDCOM.AppServer.GetOrderCount(OrdCount);
  laOrdCount.Caption:=IntToStr(OrdCount);
  conDCOM.Close;
 finally
  Screen.Cursor := crDefault;
 end
end;
```

При нажатии кнопки компонент `TDCOMConnection` устанавливает соединение с базой данных через созданный нами ранее сервер.

На втором этапе используется свойство `AppServer` компонента `TDCOMConnection`. Это свойство представляет собой ссылку на интерфейс `IAppServer` удаленного модуля данных MTS сервера. Помимо собственных методов этот интерфейс обеспечивает доступ ко всем методам интерфейса `IDMTest` удаленного модуля данных. В частности, нас интересует метод `GetOrderCount`.

Этот метод возвращает в параметре `OrdCount` целое число, означающее количество запросов в базе данных (см. листинг 4.3). После преобразования типов поместим это число в компонент `TLabel`.

На последнем этапе соединение с базой данных закрывается.

После компиляции приложение можно тестировать.

4.5.3. Пример создания клиента и сервера в случае распределенной транзакции

При распределенной транзакции в процессе должны участвовать уже несколько объектов MTS.

В нашем случае ими будут два удаленных модуля данных MTS. При этом не важно, к какой именно базе данных обращаются эти модули данных — главное, чтобы распределенная транзакция использовала различные объекты MTS. Поэтому для простоты во втором соединении используем ту же, что в первом соединении, базу данных.

Создание серверов

В этом примере мы выполним соединение с базой данных при помощи BDE. В папке ..\Program Files\Common Files\Borland Shared\Data есть набор файлов для базы данных DBDEMOS в формате dBASE. Эта база данных аналогична использованной в предыдущем примере.

Для первого модуля данных (назовем его `DMTest1`) возьмем за основу предыдущий пример и используем запрос к базе данных, возвращающий общее число заказов. Структура этого проекта полностью совпадает с только что рассмотренным примером, за исключением компонентов доступа к данным.

Для организации доступа к базе данных с помощью BDE применим стандартный псевдоним BDE DBDEMOS. Он создается и настраивается автоматически при инсталляции Delphi. Поскольку мы используем только один компонент запроса, параметры соединения достаточно настроить в самом компоненте `TQuery`, не применяя специализированный компонент соединения BDE. Для этого в списке свойства `DatabaseName` компонента `TQuery` выберем имя псевдонима DBDEMOS. После этого в свойство `SQL` введем текст запроса.

Откомпилируем проект и зарегистрируем удаленный модуль данных в пакете MTS TestPack (см. рис. 4.7, 4.8).

Создадим еще один проект удаленного модуля данных MTS. Назовем его `DMTest2`. Его структура полностью аналогична модулю данных `DMTest1`, за исключением того, что запрос возвращает общее число заказчиков.

Откомпилируем второй проект и добавим новый модуль данных в тот же пакет TestPack.

Примечание

Для удобства работы с проектами в этом примере как нельзя лучше подходит группа проектов.

Разработка объекта распределенной транзакции

Для реализации распределенной транзакции необходимо создать дополнительный объект, управляющий распределенными транзакциями.

Для этого при помощи Мастера создания объекта транзакции создадим новый объект MTS (листинг 4.4).

Листинг 4.4. Модуль объекта распределенной транзакции

```
unit uDistObject;
{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  ActiveX, Mtsobj, Mtx, ComObj, DistrObject_TLB, MTSServer1_TLB,
  MTSServer2_TLB, StdVcl;

type
  TDistrObject = class(TMtsAutoObject, IDistrObject)
  private
    DMTest1: IDMTest1;
    DMTest2: IDMTest2;
  protected
    procedure ExecTransaction(out OrdCount, CustCount: Integer); safecall;
  end;

implementation
```

```
uses ComServ;

procedure TDistrObject.ExecTransaction(out OrdCount, CustCount: Integer);
var Tr: ITransactionContextEx;
begin
 Tr := CreateTransactionContextEx;
 try
  OleCheck(Tr.CreateInstance(CLASS_DMTest1,IDMTest1, DMTest1));
  OleCheck(Tr.CreateInstance(CLASS_DMTest2,IDMTest2, DMTest2));
  DMTest1.GetOrderCount(OrdCount);
  DMTest2.GetCustCount(CustCount);
  Tr.Commit;
 except
  Tr.Abort;
 end;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TDistrObject, Class_DistrObject,
    ciMultiInstance, tmBoth);
end.
```

В объекте транзакции `TDistrObject` нам необходимо использовать два созданных ранее сервера. Для этого в секции `private` опишем две переменные, указывающие на интерфейсы этих серверов. При этом в секции `uses` должны быть подключены соответствующие библиотеки типов.

Затем для объекта `TDistrObject` разработаем обычным способом метод `ExecTransaction`, который должен возвращать результаты работы двух серверов.

В начале метода мы создаем экземпляр интерфейса `ITransactionContextEx`, который включает в себя возможности контекста распределенной транзакции. Этот интерфейс и выполняет всю работу.

Метод `CreateInstance` обеспечивает создание экземпляров интерфейсов серверов. В качестве параметров метода передаются GUID серверов и переменные интерфейсов.

Затем при помощи методов `GetOrderCount` и `GetCustCount` мы получаем информацию от двух серверов.

Метод `Commit` завершает транзакцию. А в случае возникновения ошибки метод `Abort` выполняет откат распределенной транзакции.

Откомпилируем проект и установим объект распределенной транзакции в тот же пакет TestPack.

Создание клиента

Клиентская часть приложения создается аналогично предыдущему примеру с небольшими изменениями.

В качестве сервера для компонента `TDCOMConnection` необходимо использовать объект распределенной транзакции `TDistrObject`. Его можно задать свойством `ServerName` *(см. предыдущий пример)* или свойством `ServerGUID`, в котором должен содержаться GUID объекта.

Соответственно, метод-обработчик кнопки `buCalculate` выглядит так:

```
procedure TfmMain.buCalculateClick(Sender: TObject);
var OrdCount, CustCount: Integer;
begin
 Screen.Cursor := crHourGlass;
 try
  conDCOM.Open;
  conDCOM.AppServer.ExecTransaction(OrdCount, CustCount);
  laOrdCount.Caption:=IntToStr(OrdCount);
  laCustCount.Caption:=IntToStr(CustCount);
  conDCOM.Close;
 finally
  Screen.Cursor := crDefault;
 end
end;
```

Возвращаемые значения передаются в комоненты `TLabel` для визуализации.

Резюме

Технология Microsoft Transaction Server основана на технологиях COM и Автоматизация. Название MTS используется для версий, работающих в операционных системах Windows 95/98 и NT. Для Windows 2000 разработан новый вариант, носящий название COM+.

Применение MTS в распределенных приложениях позволяет повысить их производительность и безопасность данных. Основным элементом технологии, используемым при создании приложений, является объект транзакции. Как правило, он отличается небольшими размерами и содержит часть бизнес-логики приложения, а среда MTS "умеет" работать с ним. При этом для каждого объекта транзакции среда MTS создает вспомогательный объект контекста транзакции.

Часть II

Технологии доступа к данным

Глава 5. Архитектура приложений баз данных

Глава 6. Технология dbExpress

Глава 7. Использование ADO средствами Delphi

Глава 5

Архитектура приложений баз данных

Приложение баз данных, как следует уже из его названия, предназначено для взаимодействия с некоторым источником данных — базой данных. Взаимодействие подразумевает выполнение следующих операций:

- получение данных из базы данных;
- представление данных в определенном формате для просмотра пользователем;
- редактирование данных в соответствии с реализованными в программе бизнес-алгоритмами;
- возврат обработанных данных 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.

Приложение включает в себя следующие части.

- *Механизм получения и отправки данных*, обеспечивающий 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*, являющийся ядром приложения баз данных. Он обеспечивает хранение полученных данных в приложении и предоставляет их по запросу других частей приложения.
- *Пользовательский интерфейс*, обеспечивающий просмотр и редактирование данных, а также управление данными и приложением в целом.
- *Бизнес-логика* приложения, представляющая собой 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, способов соединения приложения и базы данных. Промежуточное ПО может быть разработано как окружение приложения, без которого приложение вообще не будет работать, как набор драйверов и динамических библиотек, к которым обращается приложение, а 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 главе мы подробно остановимся на общих методах разработки приложений баз данных в Delphi. Несмотря на разнообразие способов реализации и обилие технических деталей, общая архитектура приложений баз данных в Delphi соответствует описанной выше общей схеме.

В Delphi 6 реализовано достаточно большое число разнообразных технологий доступа к данным (те из них, которые применимы к распределенным и Internet-приложениям, рассматриваются далее в этой книге). Но при этом последовательность операций при конструировании приложений баз данных остается почти одинаковой. Используются 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, ADO или dbExpress.

Итак, в этой главе обсуждаются следующие темы:

- структура приложения баз данных в Delphi;
- базовые компоненты, используемые при разработке приложений баз данных, и их взаимосвязь;

- модуль данных;
- программная реализация частей приложения баз данных;
- понятие набора данных и его участие в основных механизмах приложения баз данных;
- виды компонентов, описывающие наборы данных;
- поля, типы полей, объекты полей;
- использование индексов и реализация некоторых методов управления наборами данных, основанных на индексах;
- параметры запросов и хранимых процедур.

5.1. Общая струкрура приложения баз данных

Итак, для начала разберемся, как в целом устроено приложение баз данных и какие компоненты и программные структуры являются его необходимыми составными частями. Сначала мы опишем структуру приложения в общем. Более детально используемые здесь понятия рассматриваются далее в этой главе.

5.1.1. Как работает приложение баз данных

В Репозитарии 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, так как приложение баз данных имеет *пользовательский интерфейс*. Этот интерфейс создается с помощью стандартных и специализированных визуальных компонентов на обычных формах.

Визуальные компоненты отображения данных расположены на странице **Data Controls** Палитры компонентов. В большинстве своем они представляют модификации стандартных элементов управления, приспособленных для работы с набором данных.

Приложение может содержать произвольное число форм и использовать 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 (их удобно представить в виде таблиц в памяти), переданных из базы данных в приложение для просмотра и редактирования. Каждый набор данных хранится 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`.

Для обеспечения связи набора данных с визуальными компонентами отображения данных используется специальный компонент `TDataSource`. Его роль заключается в управлении потоками данных между набором данных и связанными с ним компонентами отображения данных. 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 на странице **Data Access** Палитры компонентов.

Таким образом, базовый механизм доступа к данным создается триадой компонентов:

- компоненты, содержащие набор данных (потомки класса `TDataSet`);
- компоненты `TdataSource`;
- визуальные компоненты отображения данных.

Рассмотрим схему взаимодействия этих компонентов в приложении БД (рис. 5.1).

В приложении с источником данных или промежуточным программным обеспечением взаимодействует компонент доступа к данным, который инкапсулирует набор данных и обращается к функциям соответствующей технологии доступа к данным для выполнения различных операций. Компонент набора данных 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 не активен, то компонент `TDataSource` обеспечивает удаление данных из компонентов отображения данных и перевод их в неактивное состояние. Или если набор данных работает в режиме "только для чтения", то компонент `TDataSource` обязан передать в компоненты отображения данных запрет на изменение данных.

С одним компонентом `TDataSource` могут быть связаны несколько визуальных компонентов отображения данных. Эти компоненты представляют собой модифицированные элементы управления, которые предназначены для показа информации из наборов данных.

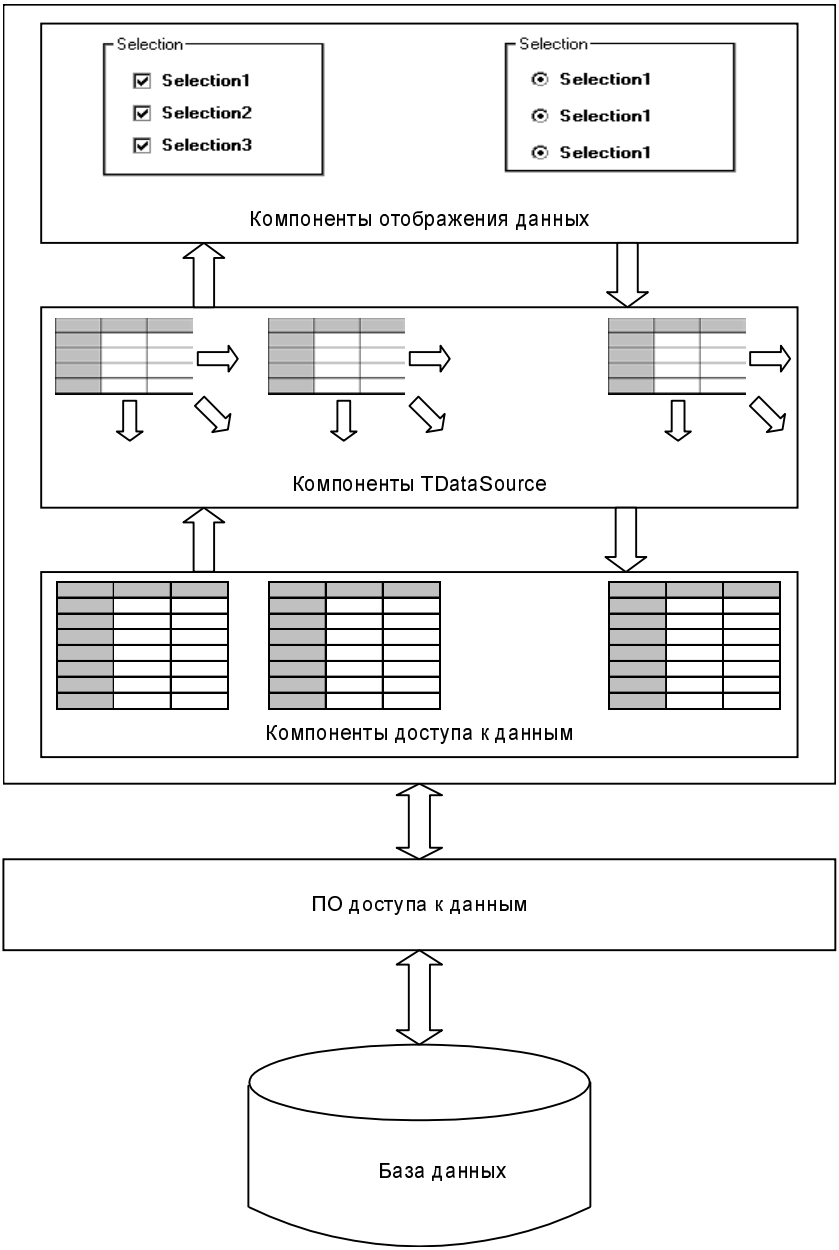


Рис. 5.1. Механизм доступа к данным приложения баз данных

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 визуальных 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.

5.1.2. Модуль данных

Для размещения компонентов доступа к данным в приложении баз данных желательно использовать специальную "форму" — модуль данных (класс `TDataModule`) (рис. 5.2). 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 программисту, как и любой другой модуль проекта, на этапе разработки. Пользователь не может увидеть модуль данных во время выполнения приложения.

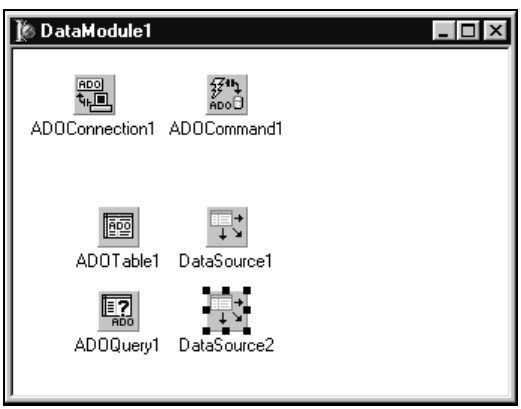


Рис. 5.2. Модуль данных

Для создания модуля данных можно воспользоваться Репозиторием объектов или главным меню Delphi. Значок модуля данных **Data Module** расположен на странице Репозитория **New**.

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 *(см. гл. 8)*.

Для создания схемы (модели, диаграммы) данных, с которой работает приложение, можно воспользоваться возможностями, предоставляемыми страницей **Diagram** Редактора кода. Любые элементы из иерархического дерева компонентов модуля данных можно перенести на страницу диаграммы и задать связи между ними.

При помощи управляющих кнопок между элементами диаграммы устанавливаются отношения синхронного просмотра и главный/подчиненный. При этом производится автоматическая настройка свойств соответствующих компонентов.

Для обращения к компонентам доступа к данным, расположенным в модуле данных, из других модулей проекта необходимо включить имя модуля в секцию `uses`:

```
...
implementation

uses uDataModule;
...
DM.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 подключен этот модуль данных. Кроме этого, все обработчики событий этих компонентов, т. е. вся логика работы с данными приложения, собраны в одном месте, что тоже весьма удобно.

5.1.3. Подключение данных

На первом этапе создания приложения баз данных необходимо связать его с источником данных. Для этого используется компонент набора данных.

Компонент набора данных является основой приложения баз данных. Для выбранной таблицы БД он создает набор данных и позволяет эффективно управлять им. В процессе работы такой компонент тесно взаимодействует с функциями соответствующей технологии доступа к данным, используя совокупность интерфейсов.

Все компоненты доступа к данным являются невизуальными. Желательно размещать их в модулях данных. Рассмотрим последовательность действий

по настройке компонентов приложения баз данных. В качестве примера используем табличный компонент.

1. Перенести компонент набора данных в модуль данных и подключить его к базе данных. Для этого, в зависимости от конкретной технологии, используется или специальный компонент, устанавливающий соединение, или прямое обращение к источнику данных, драйверу, интерфейсу или динамической библиотеке. *Детально способы соединения рассматриваются в следующих главах.*
2. Подключить к компоненту таблицу БД. Для этого используется свойство `TableName` компонента, доступное в Инспекторе объектов (Object Inspector). 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, 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 используется 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.

5.1.4. 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 используется свойство `DataSet` компонента `TDataSource`, доступное в Инспекторе объектов. Это указатель на экземпляр компонента набора данных. 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 с ними наборов данных. Обычно название компонента комбинирует имя набора данных с каким-либо префиксом (например, `OrdSource` или `dsOrders`).

Примечание

Компонент `TDataSource` можно подключить не только к набору данных из той же формы, но и из любой другой, модуль которой указан в секции `uses`.

Компонент `TDataSource` имеет ряд полезных свойств и методов.

Связь с компонентом набора данных устанавливается свойством

```
property DataSet: TDataSet;
```

а определить текущее состояние набора данных можно, используя 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. Например, если необходимо выполнить ускоренную навигацию по записям большого набора данных, все связанные с ним визуальные компоненты можно отключить.

Свойство

```
property AutoEdit: Boolean;
```

если его значение равно `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.

5.1.5. Отображение данных

На третьем этапе создания приложения баз данных необходимо разработать пользовательский интерфейс на основе *компонентов отображения данных*. Эти компоненты предназначены специально для просмотра и редактирования данных. Внешне большинство этих компонентов ничем не отличаются от стандартных элементов управления. Более того, многие из компонентов отображения данных 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

```
property DataSource: TDataSource;
```

Оно присутствует во всех компонентах отображения данных.

Большинство компонентов предназначены для представления данных из одного единственного поля. В таких компонентах имеется еще одно свойство

```
property DataField: String;
```

которое определяет поле связанного набора данных, отображаемое в компоненте.

Таким образом, для каждого визуального компонента отображения данных необходимо выполнить следующие действия.

1. Связать компонент отображения данных и компонент `TDataSource`. Для этого используется 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 используется 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.

5.2. Набор данных

Множество записей одной или нескольких таблиц, переданное 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. Этот класс должен описывать набор данных и обладать методами для управления записями и полями.

Класс `TDataSet` является базовым классом иерархии, он инкапсулирует абстрактный набор данных и реализует 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.

На основе базового класса реализованы специальные компоненты VCL для различных технологий доступа к данным, которые позволяют разработчику конструировать приложения баз данных, используя одинаковые приемы и настраивая одни и те же свойства. Таким образом, изучив возможности класса TDataSet, мы получим сведения, одинаково хорошо применимые для любых компонентов, инкапсулирующих набор данных.

5.2.1. 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. Однако он может использоваться разработчиками в качестве основы для создания собственных компонентов.

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;
```

Навигация по набору данных

В открытом наборе данных можно перемещаться по записям. Текущая запись соответствует понятию *курсора* в серверах БД. После открытия набор данных устанавливает курсор на первую запись.

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 фнукцию 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 параметр 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 использовать его нужно аккуратно, так как 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 используемая операция), можно использовать метод

```
function IsEmpty: Boolean;
```

который возвращает `True`, если набор данных пуст. Кроме того, можно воспользоваться уже упоминавшимися свойствами

```
...
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;
```

Поля набора данных

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 необязательно набор данных должен содержать все поля таблицы базы данных.

Совокупность полей набора данных хранит 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, следует использовать метод

```
function FieldByName(const FieldName: string): TField;
```

И делается это так:

```
MyTable.FieldByName('VENDORNO').AsInteger := 1234;
```

Имя поля, передаваемое в параметре `FieldName`, не чувствительно к регистру символов.

Метод

```
procedure GetFieldNames(List: TStrings);
```

вернет в параметре `List` полный список имен полей набора данных.

Более подробная информация о полях и способах работы с ними обсуждается в следующих разделах.

Редактирование набора данных

Класс `TDataSet` содержит ряд свойств и методов, которые обеспечивают редактирование набора данных.

Сначала полезно узнать, можно ли редактировать набор данных вообще с помощью свойства

```
property CanModify: Boolean;
```

которое принимает значение `True` для редактируемых наборов.

Перед началом редактирования набор данных нужно перевести в режим редактирования, используя метод

```
procedure Edit;
```

Для сохранения сделанных изменений применяется метод

```
procedure Post; virtual;
```

Метод `Post` вызывается либо разработчиком, либо самим набором данных при переходе на другую запись.

При необходимости все сделанные после последнего вызова `Edit` 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 с уже заполненными полями. Для этого применяются методы

```
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
```

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.

Для существующей записи аналогичным образом можно заполнить все поля, используя метод

```
procedure SetFields(const Values: array of const);
```

Текущая запись удаляется методом

```
procedure Delete;
```

При этом набор данных не выдает никаких предупреждений.

Очистить содержимое всех полей текущей записи может метод

```
procedure ClearFields;
```

Обратите внимание, что поля становятся пустыми (`NULL`), а не приобретают нулевые значения.

О том, что редактировалась 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 обновляет только те таблицы и запросы, которые нельзя редактировать.

Обработка событий набора данных

Методы-обработчики класса `TDataSet` предоставляют программисту широчайшие возможности для отслеживания 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 на то, что помимо методов-обработчиков режима вставки существует еще один метод

```
property OnNewRecord: TDataSetNotifyEvent;
```

который вызывается непосредственно при вставке или добавлении записи.

Кроме того, существуют 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 быть 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 выше обработчики имеют одинаковый тип

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
```

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`.

5.2.2. Стандартные компоненты

Если читатель проинспектирует компоненты наборов данных на страницах Палитры компонентов, он обратит внимание на то, что 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.

Имеет смысл рассмотреть общие для компонентов свойства и методы, предположив, что существуют некие виртуальные общие предки для таблицы, запроса и хранимой процедуры.

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 благодаря применению табличных индексов.

Но в практике программирования работа с таблицами целиком используется не так часто. При работе с серверами баз данных промежуточное ПО используемых технологий доступа к данным все равно транслирует запрос на получение табличного набора данных в простейший запрос SQL, например:

```
SELECT * FROM Orders
```

В такой ситуации использование табличных компонентов становится менее эффективным, чем использование запросов.

После соединения с источником данных необходимо задать имя таблицы в свойстве

```
property TableName: String;
```

Иногда в свойстве `TableType` дополнительно задается тип таблицы (например, таблицы Paradox).

Если соединение с источником данных настроено правильно, имя таблицы можно выбрать из раскрывающегося списка свойства `TableName` в Инспекторе объектов.

Преимуществом табличного компонента является использование индексов, которые ускоряют работу с таблицей. Все индексы, созданные в базе данных для таблицы, автоматически загружаются в компонент. Их параметры содержит 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 задавать составные индексы.

Свойства `IndexName` и `IndexFieldNames` нельзя применять одновременно.

Число полей, используемых 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, а также описание полей и индексов из свойств `TFieldDefs` и `TIndexDefs`. Если таблица с таким именем уже есть 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.

Дополнительную информацию о табличных компонентах см. в разд. 5.5 этой главы.

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 можно использовать только для просмотра, и конечно, возможности компонентов здесь не 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: структура возвращаемых запросом полей может изменяться, и экземпляры класса `TFieldDef`, хранящие информацию о свойствах полей, создаются лишь в случае необходимости при запуске приложения.

Примечание

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 несколькими способами.

1. 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`.

2. Если запрос не возвращает результат в набор данных (например, использует операторы `INSERT`, `DELETE`, `UPDATE`), то используется 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 нельзя редактировать.

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, так как компонент делает это автоматически. Однако если запрос будет выполняться несколько раз подряд, то подготовку необходимо провести перед первым выполнением запроса вручную. Тогда при последующих выполнениях

запроса сервер не будет тратить время на проведение бесполезной операции — ведь ресурсы для запроса уже были выделены.

Часто запросы имеют настраиваемые параметры, значения которых определяется 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` и имеет тип `TParameters`.

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 выполняется.

Кроме этого, компоненты запросов содержат некоторые свойства и методы, описываемые в разделе "Механизмы управления данными" этой главы.

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.

Общее число параметров возвращает свойство

```
property ParamCount: Word;
```

Для подготовки хранимой процедуры используется метод

```
procedure Prepare;
```

или свойство

```
property Prepared: Boolean;
```

которое должно получить значение `True`.

Метод

```
procedure UnPrepare;
```

или свойство `Prepared := False` выполняют обратное действие.

Кроме того, проверка значения свойства `Prepared` позволяет установить, выполнялась ли подготовка процедуры к выполнению.

Внимание!

После выполнения хранимой процедуры исходный порядок следования параметров в списке `Params` может измениться. Поэтому для доступа к конкретному параметру рекомендуется использовать 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 используется метод

```
procedure ExecProc;
```

и после этого выходные параметры получат вычисленные значения.

5.2.3. 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.

- К первой группе относятся состояния, в которые набор данных переходит автоматически: это непродолжительные по времени состояния, сопровождающие функционирование полей набора данных (табл. 5.1).
- Во вторую группу входят состояния, которыми можно управлять из приложения, например, перевод набора данных в режим редактирования (табл. 5.2).

Базовый класс `TDataSet`, инкапсулирующий свойства набора данных, позволяет изменять его 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 используются методы `Open`, `Close`, `Edit`, `Insert`.

Таблица 5.1. *Автоматические состояния набора данных*

Константа состояния	Описание
`dsNewValue`	Устанавливается при обращении к свойству `NewValue` поля набора данных
`dsOldValue`	Устанавливается при обращении к свойству `OldValue` поля набора данных
`dsCurValue`	Устанавливается при обращении к свойству `CurValue` поля набора данных
`dsInternalCalc`	Устанавливается при расчете значений полей, для которых `FindKind = fkInternalCalc`
`dsCalcFields`	Устанавливается при выполнении метода `OnCalcFields`
`dsBlockRead`	Устанавливается при использовании механизма ускоренного перемещения по набору данных
`dsOpening`	Существует при открытии набора данных методом `Open` или свойством `Active`
`dsFilter`	Устанавливается при выполнении метода `OnFilterRecord`

Таблица 5.2. *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`	Устанавливается 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 из режима просмотра данных (рис. 5.3).

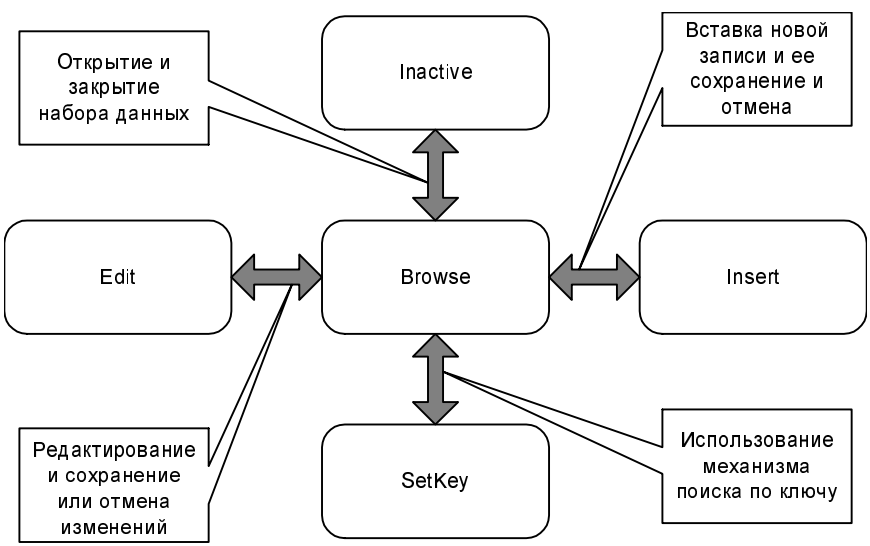


Рис. 5.3. Схема изменения состояний набора данных

При необходимости редактирования данных набор должен быть переведен в состояние редактирования `dsEdit`, для этого используется 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 использовать 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` используется 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` используется набором данных при реализации быстрого перемещения по большим массивам записей без показа промежуточных записей в компонентах отображения данных и без вызова обработчика события перемещения по записям. Для реализации быстрого перемещения по набору данных можно использовать методы `DisableControls` и `EnableControls`.

5.3. Индексы

Важнейшей проблемой для любой БД является достижение максимальной производительности и ее сохранение при дальнейшем увеличении объемов хранимых данных. Использование индексов позволяет 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 тех таблиц, в которых 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 индекс "знает", как организованы данные, и может ускорять обработку за счет группирования записей по сходным значениям параметров.

Естественно, что в компонентах доступа к данным VCL Delphi используются все возможности такого мощного инструмента, как индексы. Причем свойства и методы для работы с индексами присутствуют только в табличных компонентах, так как 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 используется первичный индекс.

5.3.1. Механизм подключения индексов

Для того чтобы подключить к набору данных вторичный индекс, необходимо присвоить свойству

```
property IndexName: String;
```

название индекса. Если свойство не имеет значения, то в наборе данных используется первичный индекс.

Альтернативный способ задания индекса заключается в использовании свойства

```
property IndexFieldNames: String;
```

в котором задается перечень имен полей необходимого индекса, разделенных точкой с запятой. При этом в Инспекторе объектов в этом свойстве для существующих индексов список полей создается автоматически, разработчику остается лишь сделать выбор. При помощи этого свойства можно создавать и составные индексы. Для этого необходимо, чтобы все входящие в список поля были индексированы.

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

```
property IndexFieldCount: Integer;
```

5.3.2. 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 с помощью свойства

```
property Items[Index: Integer]: TIndexDef; default;
```

являющегося 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.

5.3.3. 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 используется сортировка записей с учетом регистра символов. Но некоторые серверы БД допускают комбинированную сортировку по полям с учетом регистра и без.

Свойство

```
property DescFields: String;
```

содержит список полей через точку с запятой, 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 индексе используется 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`.

Например, свойства и методы описаний индексов можно использовать так:

```
with Table1 do
begin
...
 with IndexDefs do
 begin
  Clear;
```

```
  AddIndexDef;
  with Items[0] do
  begin
   Name := 'Primary';
   Fields := 'Field1';
   Options := [ixPrimary, ixUnique];
  end;
  AddIndexDef;
  with Items[1] do
  begin
   Name := 'SecondIndex';
   Fields := 'Field1;Field2';
   Options := [ixCaseInsensitive];
  end;
 end;
end;
```

При создании описаний индексов использован метод `AddIndexDef`, который при каждом вызове добавляет к списку `Items` объекта `TIndexDefs` новый объект `TIndexDef`. Сначала создается первичный индекс, затем вторичный индекс `SecondIndex`. Для каждого описания обязательно определяются составляющие индекс поля и параметры индекса (свойства `Fields` и `Options`).

5.4. Параметры запросов и хранимых процедур

Свойство

```
property Params: TParams;
```

представляет собой набор изменяемых параметров запроса или хранимой процедуры. Свойство представляет собой набор объектов `TParam`, содержащих отдельные параметры.

Рассмотрим следующий запрос SQL:

```
SELECT SaleDat, OrderNo
FROM Orders
WHERE SaleDat>= '01.08.2002' AND SaleDat<='31.08.2002'
```

В нем осуществляется отбор из таблицы (`Orders`) номеров заказов (`OrderNo`), сделанных в августе 2002 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:

```
...
with Query1 do
begin
 Close;
 SQL.Clear;
 SQL[0] := 'SELECT PartDat, ItemNo, ItemCount, InputPrice';
 SQL[1] := 'FROM Parts';
 SQL[2] := 'WHERE PartDat>= '+chr(39)+Date1Edit.Text+chr(39)+
  ' AND PartDat<='+chr(39)+Date2Edit.Text+chr(39);
  Open;
end;
```

Здесь используется свойство `SQL`, и в соответствии с заданными в однострочных редакторах `Date1Edit` и `Date2Edit` датами изменяется текст запроса.

Для решения подобных задач как раз и используются 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 по тексту 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. 5.4). Для каждого параметра требуется установить тип данных, который должен соответствовать типу данных соответствующего поля.

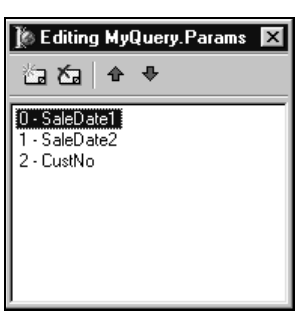


Рис. 5.4. Специализированный редактор параметров запроса

Теперь для задания текущего диапазона дат поступления можно использовать свойство Params:

```
...
with Query1 do
begin
 Close;
 Params[0].AsDateTime := StrToDate(Date1Edit.Text);
 Params[1].AsDateTime := StrToDate(Date2Edit.Text);
 Open;
end;
...
```

При помощи параметров в запрос передается текущий диапазон дат.

Значения параметров запроса можно задать и из другого набора данных. Для этого применяется 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 с параметрами полей автоматически передаются в запрос.

Например, свойство SQL в компоненте первого запроса выглядит так:

```
SELECT OrderNo, CustNo, SaleDate, AmountPaid FROM Orders
```

Он должен быть связан с компонентом типа TDataSource.

Свойство SQL второго запроса имеет следующий вид:

```
SELECT CustNo, Company FROM Customer
WHERE CustNo=:CustNo
```

Обратите внимание, что название параметра соответствует названию поля номера заказчика (CustNo) из таблицы заказов (Orders). Это условие обязательно. Свойство DataSource компонента второго запроса должно указывать на компонент TDataSource первого запроса. Как только оба набора данных открываются, текущее значение из поля CustNo набора данных Orders автоматически передается в параметр компонента запроса набора данных Customers (Заказчики). Благодаря этому для двух наборов данных реализована связь "один ко многим" по полю номера заказчика.

И в завершение разговора о параметрах запросов рассмотрим свойства и методы класса TParams, используемого для определения свойства Params, и класса TParam, описывающего отдельный параметр.

5.4.1. 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;
```

где `FldType` — тип данных параметра, `ParamName` — имя параметра и `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 использовать метод

```
function ParseSQL(SQL: String; DoCreate: Boolean): String;
```

который при `DoCreate = True` анализирует текст запроса из `SQL` и создает новый список параметров.

Или же, для присвоения значений сразу всем параметрам используется метод

```
procedure AssignValues(Value: TParams);
```

Для удаления параметра из списка применяется метод

```
procedure RemoveParam(Value: TParam);
```

Для идентификации параметров полезно использовать обращение к ним по имени, так как при работе с хранимыми процедурами, после их выполнения порядок следования параметров может измениться. То же самое может произойти и при использовании динамических запросов (их текст SQL может изменяться во время выполнения).

Для обращения к параметру по имени применяется метод

```
function ParamByName(const Value: String): TParam;
```

В сложных запросах SQL или после многочисленных исправлений запроса 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 используется метод

```
function IsEqual(Value: TParams): Boolean;
```

который возвращает `True`, если для параметра `Value` найден дубликат.

5.4.2. 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 с ним 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 приложение информации о статусе операции.

А свойство

```
property ParamType: TParamType;
```

определяет тип параметра.

При работе с параметрами довольно часто необходимо определить, имеет ли параметр ненулевое значение. Для этого используется свойство

```
property IsNull: Boolean;
```

Свойство возвращает `True`, если параметр не имеет значения или имеет значение `Null`.

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;
```

Использование типа `Variant` не очень эффективно, так как не обеспечивает максимальной скорости работы набора данных. 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.

Замечание

Необходимо осторожно использовать свойства с приведением типа, так как попытка преобразования неверного значения вызовет исключительную ситуацию.

Для чтения из буфера и записи в буфер значения параметра соответственно используются 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 используется метод

```
procedure AssignFieldValue(Field: TField; const Value: Variant);
```

Общую разрядность 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;
```

5.5. Механизмы управления данными

5.5.1. Связанные таблицы

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 использовать любой компонент, инкапсулирующий набор данных. Для задания подчиненной таблицы можно использовать только табличные компоненты.

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 выполняться только при перемещениях по записям главной таблицы.

Свойство

```
property MasterSource: TDataSource;
```

определяет компонент `TDataSource`, который связан с главной таблицей.

Затем при помощи свойства

```
property MasterFields: String;
```

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. 5.5).

Здесь в раскрывающемся списке **Available Indexes** выбирается требуемый индекс для подчиненной таблицы. Однако этот список появляется не для всех технологий доступа к данным.

После этого в списке **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 между двумя полями главной и подчиненной таблиц. Созданная связь отображается в списке **Joined Fields**.

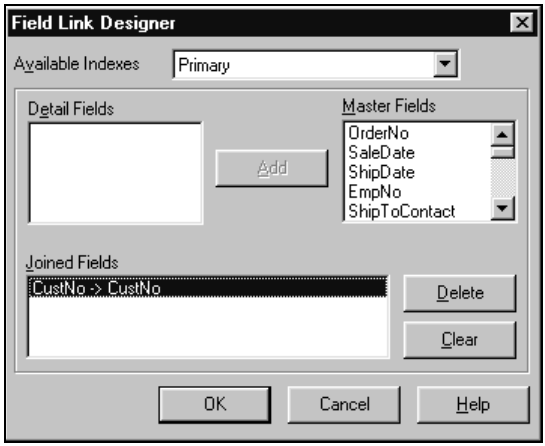


Рис. 5.5. Редактор связей полей

Примечание

После создания связи по индексированным полям данный индекс становится текущим для набора данных. При этом в зависимости от типа СУБД автоматически заполняется свойство `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" установлено. Теперь достаточно открыть оба набора данных, чтобы увидеть работу отношения.

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".

5.6. Поиск данных

Для поиска в произвольной выборке полей можно использовать 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 значение `loCaseInsensitive`, которое отключает проверку на соответствие регистру символов и значение `loPartialKey`, которое включает поиск с минимальными отличиями. В случае успешного поиска курсор набора данных устанавливается на найденной записи, а метод возвращает значение `True`. Пример вызова метода:

```
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 `Variant` для полей, названия которых содержатся в параметре `ResultFields`. Пример обращения к методу:

```
Table1.Lookup('Last_Name;First_Name', VarArrayOf(['Edit1.Text',
'Edit2.Text']), 'Last_Name;First_Name');
```

Оба эти метода автоматически используют быстрый индексный поиск, если в параметре `KeyFields` задать поля индекса.

5.7. 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 запроса также может использовать фильтры. Эта возможность подчас позволяет легко и изящно решать довольно сложные проблемы, которые в противном случае требуют изменения текста запроса или создания нового компонента запроса.

При использовании фильтра его текст транслируется в синтаксис языка 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 применять как свойство `Filter`, так и метод `OnFilterRecord`.

Динамические фильтры можно создавать и редактировать во время выполнения приложения, для них используется только свойство `Filter`.

При вводе текста фильтра в свойство `Filter` используются имена полей соответствующей таблицы БД, а для задания отношений применяются все операторы сравнения (>, >=, <, <=, =, <>) и логические операторы `AND`, `OR`, `NOT`, например:

```
Field1>100 AND Field2=20
```

Сравнивать между собой два поля нельзя. Следующий фильтр вызовет ошибку при попытке использования:

```
ItemCount=Balance AND InputPrice>OutputPrice
```

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 используется звездочка:

```
ItemName='A*'
```

Фильтр начинает работать только после того, как свойству `Filtered` присваивается значение `TRUE`. 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;
```

Если в свойстве указано значение `foCaseInsensitive`, то отключается сравнение строковых значений с учетом регистра символов. Значение `foNoPartialCompare` включает отбор строковых значений по частичному совпадению строк.

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 нет:

```
procedure TForm1.TableFilterRecord(DataSet: TDataSet; var Accept:
Boolean);
begin
 Accept := Orders.FieldByName['SaleDate'].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 с их значениями.

К сожалению, метод обладает недостаточной гибкостью, хотя такой фильтр можно модифицировать путем присвоения методу значения процедурной переменной, содержащей ссылку на новый метод.

5.8. 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.

Закладка представляет собой указатель на запись набора данных:

```
type TBookmark = Pointer;
```

В наборе данных есть свойство

```
type TBookmarkStr: string;
property Bookmark: TBookmarkStr;Bookmark,
```

которое содержит название текущей закладки.

При работе с закладками используются три основные метода.

Метод

```
function GetBookmark: TBookmark; virtual;
```

создает новую закладку для текущей записи.

Метод

```
procedure GotoBookmark(Bookmark: TBookmark);
```

осуществляет переход к закладке, переданной в параметре.

Метод

```
procedure FreeBookmark(Bookmark: TBookmark); virtual;
```

удаляет закладку, переданную в параметре.

Кроме этого можно применять метод

```
function BookmarkValid(Bookmark: TBookmark): Boolean; override;
```

который проверяет, указывает ли закладка на реально существующую запись.

Метод

```
function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
override;
```

позволяет сравнить между собой две закладки:

```
var Bookmark1, Bookmark2: TBookmark;
...
if Table1.CompareBookmark(Bookmark1, Bookmark2) = 1
then ShowMessage('Закладки одинаковы');
```

Примечание

Закладки также используются в компоненте `TDBGrid`. Он имеет свойство `SelectedRows` типа `TBookmarkList`, которое представляет собой список закладок, указывающих на одновременно выделенные записи.

5.9. Поля

Каждая таблица БД и, следовательно, каждый набор данных приложения имеет собственную структуру, которая определяется совокупностью полей. Каждое *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, расположенных 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 содержит основные свойства абстрактного поля, не зависящего от типа данных. От этого базового класса порождены другие классы, обеспечивающие функционирование реальных объектов полей, имеющих конкретный тип данных.

5.10. Объекты полей

Объекты полей включают в себя 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 компонента набора данных

```
property Fields: TFields;
```

представляет собой индексированный список объектов полей набора данных. Если разработчик не изменяет порядок следования полей в наборе данных, то расположение объектов полей в списке `Fields` соответствует структуре таблицы базы данных.

Каждый объект поля 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 уже приведенными способами или, если тип данных поля неизвестен, свойством `FieldValues`:

```
Table1.FieldValues['SomeField'] := Edit1.Text;
```

Знание имени поля дает самый простой способ обращения к текущему значению поля:

```
Table1['SomeField'] := Edit1.Text;
Edit1.Text := Table1['SomeField'];
```

Замечание

При присваивании значений полям набора данных необходимо контролировать состояние, в котором находится набор данных.

В основе классов, описывающих иерархию типизированных полей, лежит класс `TField`. От него порождены другие классы, обеспечивающие работу целых групп полей, объединенных по типам данных.

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).

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.

5.10.1. 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 использует только динамические поля. К свойствам и методам динамических полей можно обратиться программно, для этого следует применять индексированное свойство `Fields` компонента доступа к данным, которое объединяет все поля набора данных, или метод `FieldByName`.

Динамические поля используются в тех 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 по компоненту набора данных на форме или командой **Fields Editor** из всплывающего меню этого компонента.

Редактор полей представляет собой простой список уже созданных статических полей, все управление осуществляется командами из всплывающего меню (рис. 5.6). 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 полей 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, создаются произвольные комбинации статических полей.

Рис. 5.6. 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, который указан 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. 5.7). Для выбора типа поля используется 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.

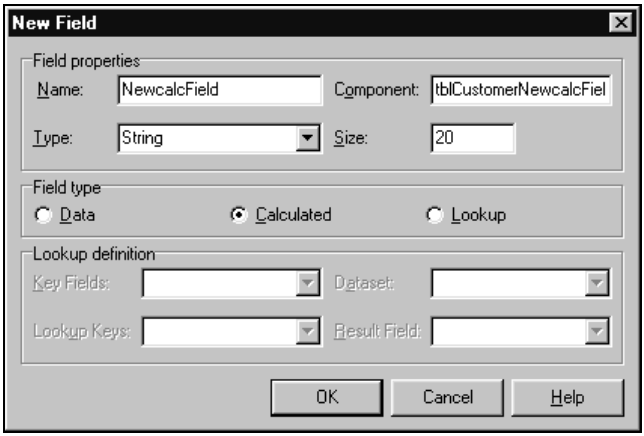


Рис. 5.7. Диалоговое окно создания нового статического поля в Редакторе полей набора данных

Примечание

Для клиентских наборов данных многоуровневых приложений диалоговое окно создания нового поля позволяет выбрать два дополнительных типа поля — это агрегатные поля (радиокнопка **Aggregate**) и внутренние вычисляемые поля (радиокнопка **InternalCalc**).

5.10.2. Класс *TField*

Как уже говорилось ранее, в большой иерархии классов для полей различных типов данных класс `TField` является базовым, он описывает 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.

Ниже 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 этом случае свойство содержит имена всех родительских полей.

Название поля в таблице базы данных содержится в свойстве

```
property Origin: String;
```

и используется только в компонентах запросов, так как в них поля могут иметь псевдонимы.

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 с помощью 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 сохраняется после каждого выполнения метода `Post` набора данных:

```
with Table1 do
begin
 Open;
 while Not EOF do
 begin
  if Fields[0].Value > 10 then
  begin
   Edit;
   Fields[1].Value := Fields[1].Value*2;
   Post;
  end;
  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 используется свойство `Value` объектов полей набора данных.

Однако применение типа `Variant` замедляет работу со свойством `Value`. Для преобразования текущего значения поля к необходимому типу данных можно использовать целую группу быстрых свойств `As...`, которые содержат

значение определенного типа данных. Чаще всего применяется свойство AsString, например, для представления числовых значений полей в элементах управления:

```
Edit1.Text := Table1.Fields[0].AsString;
```

Примечание

При передаче значений статическими объектами полей желательно использовать свойства из группы As..., так как неявное задание типа свойством Value может привести к ошибке преобразования данных типа Variant.

Если свойство

```
property CanModify: Boolean;
```

имеет значение False, значение 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 следующие значения:

```
DisplayText = '1 452,32'
Text = '1452,32'
```

Свойства Text и DisplayText влияют на использование метода-обработчика OnGetText.

```
type TFieldGetTextEvent = procedure(Sender: TField; var Text: String;
```

```
DisplayText: Boolean) of object;
property OnGetText: TFieldGetTextEvent;
```

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 значение содержит какие-либо символы кроме цифр, то все выражение нужно обязательно брать в кавычки.

В случае возникновения исключительных ситуаций во время использования поля 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 представления значений поля в визуальных компонентах отображения данных.

Свойство

```
property Visible: Boolean;
```

отвечает за видимость поля в визуальных компонентах отображения данных. Если значение свойства равно `False`, то 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 в следующих разделах этой главы.

5.10.3. Виды полей

Теперь рассмотрим классификацию полей набора данных в зависимости от их функционального назначения. Самыми распространенными полями 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 видов полей набора данных практически не отличается. Тем не менее, такое разнообразие позволяет успешно решать самые сложные задачи программирования приложений БД.

Мы рассмотрим только поля синхронного просмотра и вычисляемые поля, так как 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 разработке приложений для работы с базами данных часто возникает необходимость в связывании двух наборов данных по ключевому полю. Например, в таблице `Orders` (хранит данные о заказах) демонстрационной базы данных DBDEMOS есть поле `CustNo`, которое содержит идентификационный номер заказчика. Под этим же номером в таблице `Customers` хранится информация о заказчике (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 заказчиков, а их имена.

Таким образом, в наборе данных заказов вместо поля номера заказчика должно появиться поле имени заказчика 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 выбор конкретного наименования производится 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 синхронными, 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.

При создании для исходного набора данных нового поля синхронного просмотра необходимо использовать следующие свойства.

Свойство

```
property LookupDataSet: TDataSet;
```

задает набор данных синхронного просмотра.

Свойство

```
property LookupResultField: String;
```

представляет поле синхронного просмотра, в созданном поле будут появляться данные из набора данных `LookupDataSet`.

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.

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 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 речь пойдет дальше, обязательно должно иметь значение `False`.

Для идентификации полей синхронного просмотра можно использовать булевское свойство `Lookup` базового класса `TField`, которое принимает истинное значение для таких полей.

Свойство

```
property LookupCache: Boolean;
```

определяет режим использования специального буфера значений синхронного просмотра. Если это свойство истинно, то буфер работает.

Буфер реализован при помощи свойства

```
property LookupList: TLookupList;
```

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 синхронного просмотра 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 использовании низкоскоростных сетей.

При изменениях в наборе данных синхронного просмотра можно применять 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 диалоговом окне (см. рис. 5.7), помимо обычных действий, соответствующих созданию поля данных, необходимо задать значения свойств в группе **Lookup definition**. Элементы управления группы становятся доступны после выбора вида 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 `LookupDataSet`). Список **Result Field** позволяет выбрать поле синхронного просмотра (свойство `LookupResultField`). Список **Lookup Keys** задает ключевое 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, так как 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 можно использовать любые

арифметические и логические операции и функции, любые операторы языка, свойства и методы любых компонентов, в том числе запросы SQL:

```
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.AsString := 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 использовании 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 диалоговом окне создания нового поля Редактора полей в качестве вида поля задать вычисляемое, в остальном процесс совпадает с созданием поля данных.

В выражениях вычисляемых полей можно использовать другие вычисляемые поля, но они обязательно должны быть определены в методе `OnCalcFields` до этого.

Вычисляемые поля нельзя использовать при фильтрации набора данных методом-обработчиком `OnFilterRecord`, так как он вызывается до метода-обработчика `OnCalcFields`, а вычисляемые поля не сохраняются.

Внутренние вычисляемые поля

Помимо простых вычисляемых полей, существуют внутренние вычисляемые поля (`FieldKind = fkInternalCalc`). Они применяются в клиентских наборах данных (компоненты `TClientDataSet`) и отличаются тем, что их значения сохраняются в наборе данных.

Внутренние вычисляемые поля могут быть использованы для фильтрации методом-обработчиком `OnFilterRecord`.

Агрегатные поля

Агрегатные поля предназначены для выполнения вычислительных операций над 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, так как 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.

Использование агрегатных полей возможно только в компоненте `TClientDataSet` и его аналогах, так как он обеспечивает кэширование данных, необходимое для проведения вычислений *(см. гл. 10)*.

Агрегатные поля не отображаются вместе со всеми полями в компонентах `TDBGrid`, в Редакторе полей они 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;
```

Для создания агрегатного поля необходимо использовать команду **New field** из всплывающего меню Редактора полей.

Объектные поля

Наряду с обычными типами данных (строковым, целочисленным и т. д.), при работе с полями набора данных можно использовать более сложные типы, представляющие собой совокупность более простых типов.

В Delphi существуют четыре класса объектных полей: `TADTField`, `TArrayField`, `TDataSetField`, `TReferenceField`. Их общим предком является класс `TObjectField`. Классы `TADTField`, `TArrayField` обеспечивают доступ из родительского поля к набору дочерних полей одного типа. Эти типы полей можно использовать, если сервер БД поддерживает объектные типы и соответствующие поля имеются в наборе данных. Объектные поля можно создавать статически и динамически, так же, как и простые поля.

Для доступа к дочерним полям в этих классах имеется свойство

```
property Fields: TFields;
```

которое представляет собой индексированный список объектов дочерних полей, и свойство

```
property FieldValues[Index: Integer]: Variant;
```

содержащее значения дочерних полей.

Свойство

```
property FieldCount: Integer;
```

возвращает количество дочерних полей.

Классы `TDataSetField` и `TReferenceField` предоставляют доступ к данным из связанных наборов данных.

Ссылка на используемый набор данных задается свойством

```
property DataSet: TDataSet;
```

5.10.4.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*, используемыми в этих базах данных.

Естественно, что существует большая группа типов данных, конкретная реализация которых практически не отличается для разных платформ, например, строки, символы, целые и вещественные числа и т. д.

Есть типы данных, которые реализованы далеко не на каждой платформе. Есть, 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, 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);
```

Примечание

В Delphi тип данных BCD напрямую не поддерживается. Его использование обеспечивает денежный тип (ftCurrency). Поэтому точность BCD ограничена 20 цифрами после запятой. Для устранения этого ограничения используется тип FMTBcd, который обладает требуемой точностью.

Наряду с традиционными типами данных, в Delphi имеются и специальные типы, использование которых значительно расширяет функциональность приложений. В частности, типы ftInterface, ftIDispatch, ftGuid позволяют создавать полноценные приложения БД для COM и OLE DB.

Практически в составе всех серверов 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, наборы данных).

Автоинкрементный тип данных ftAutoInc 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 используется в качестве первичного ключа.

Данные типа BLOB (Binary Large OBject) ftBlob 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 данных, которые имеют скалярную и нескалярную структуру и которые можно преобразовать в двоичное представление.

Тип Memo ftMemo представляет собой набор строк произвольной длины (его разновидность — форматированный Memo), он основан на формате BLOB. Этот тип используется, если необходимо сохранить текст из компонента TMemo или из текстового редактора.

Графический тип данных применяется 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 использования возможностей СУБД Paradox и dBASE при 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 используется свойство `SparseArrays` в классе `TDataSet`.

В качестве отдельного поля в набор данных можно включить и любой другой произвольный набор данных. Для этого применяется специальный тип данных `ftDataSet` и класс `TDataSetField`. Причем каждым полем из интегрированного набора данных также можно управлять.

Ссылочный тип данных `ftReference` использует внешние наборы данных, но в данном случае можно подключать и обрабатывать только отдельные поля.

Резюме

Приложения баз данных могут получать доступ к источникам данных при помощи разнообразных технологий доступа, многие из которых используются 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 являются единой основой, на которой базируются технологии доступа к данным. Это позволило унифицировать процесс проектирования приложений баз данных.

В основе процесса разработки лежит триада компонентов:

- невизуальные компоненты набора данных;
- невизуальные компоненты `TDataSource`;
- визуальные компоненты отображения данных.

Глава 6

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 около 15 Мбайт дискового пространства, а также специальная настройка псевдонимов. ADO (технология Microsoft ActiveX Data Objects) предустановлена в операционной системе, но нуждается в настраиваемых провайдерах данных *(см. гл. 7)*. А в случае обновления версии ADO, дистрибутив вашего приложения "потяжелеет" более чем на 2 Мбайта.

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 компонентов на одноименной странице.

Технология доступа к данным dbExpress обеспечивает взаимодействие приложения с серверами баз данных на основе использования 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 нет возможности прямого редактирования наборов данных.

Примечание

Проблема редактирования данных в dbExpress может быть решена несколькими путями *(см. разд. 6.7 данной главы)*. 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 четырех серверов баз данных.

- DB2
- InterBase
- MySQL
- Oracle

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.

Кроме того, технология dbExpress обеспечивает доступ к данным в кроссплатформенных приложениях для Windows и Linux, так как используется и в Delphi и Kylix, а способы ее применения идентичны.

Таким образом, технология dbExpress является наилучшим решением для приложений, в которых необходим быстрый и необременительный просмотр данных серверов SQL. Она 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.
- Способы использования компонентов dbExpress для просмотра данных и создание пользовательского интерфейса приложений.
- Программная реализация редактирования данных.
- Работа с данными в режиме кэширования изменений и использование компонента `TSQLClientDataSet`.
- Применение интерфейсов.
- Распространение приложений с интегрированной технологией dbExpress.

6.1. Доступ к данным dbExpress

Организацию доступа к данным в рамках dbExpress обеспечивают три составные части (рис. 6.1).

- Клиентское ПО сервера БД, к которому осуществляется доступ.
- Драйвер dbExpress.
- Компоненты dbExpress в составе приложения, использующего сервер БД.

Для организации доступа к данным в первую очередь на компьютере должно быть установлено и настроено клиентское ПО соответствующего сервера БД. У каждого из четырех поддерживаемых в dbExpress серверов есть особенности установки, обсуждение которых выходит за рамки данной книги.

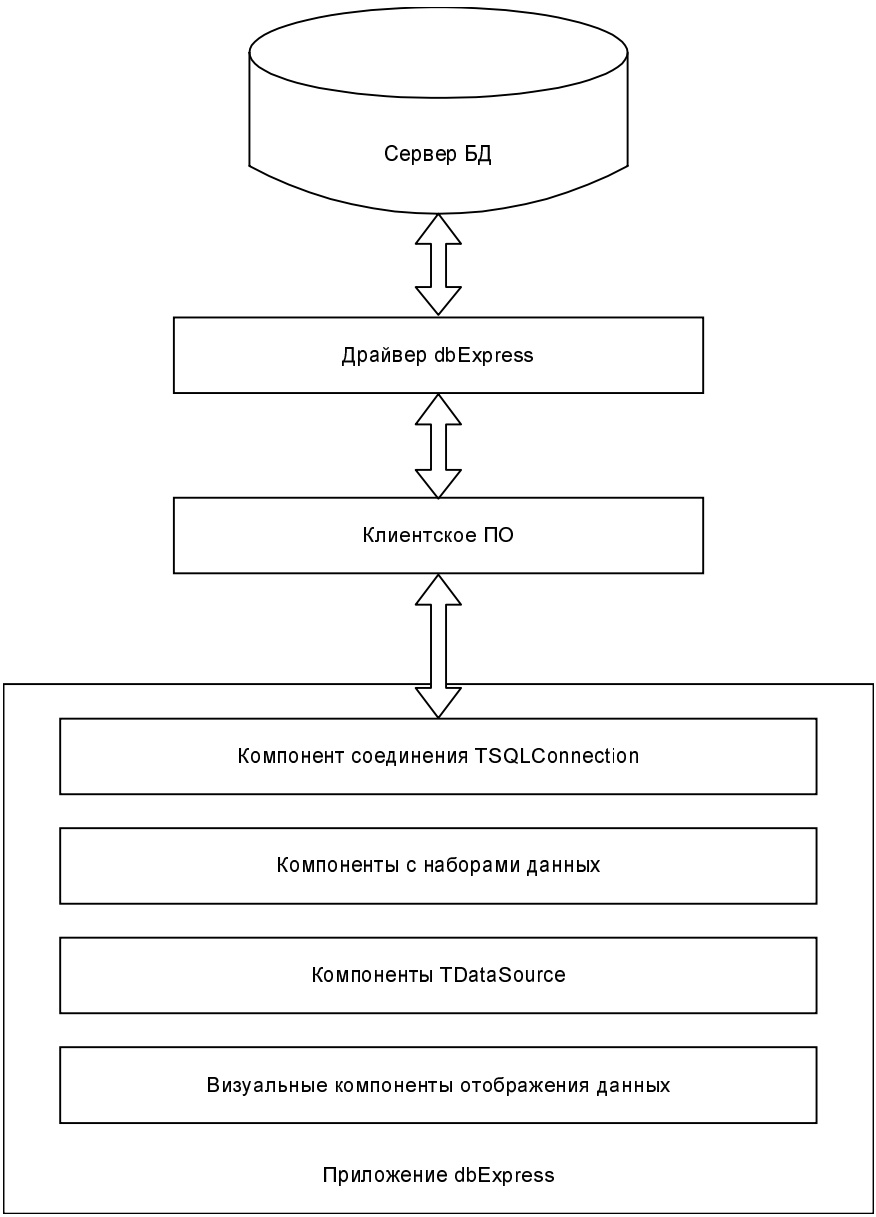


Рис. 6.1. Схема доступа к данным в рамках технологии dbExpress

Для взаимодействия с клиентским ПО сервера технология dbExpress использует специальные драйверы, которые выполняют передачу запросов от приложения.

Приложение должно содержать как минимум один компонент, обеспечивающий соединение с сервером, и необходимое количество компонентов, инкапсулирующих набор данных. В остальном, приложение не отличается от обычного приложения БД.

Все необходимые компоненты расположены на странице **dbExpress** Палитры компонентов.

6.2. Драйверы доступа к данным

Технология dbExpress обеспечивает доступ к серверу баз данных при помощи драйвера, реализованного как динамическая библиотека. Для каждого сервера имеется своя динамическая библиотека.

Перечисленные в табл. 6.1 файлы находятся в папке ..\Delphi6\Bin.

Таблица 6.1. *Драйверы dbExpress*

Сервер БД	Драйвер	Клиентское ПО
DB2	Dbexpdb2.dll	db2cli.dll
InterBase	Dbexpint.dll	GDS32.DLL
MySQL	Dbexpmy.dll	LIBMYSQL.DLL
Oracle	Dbexpora.dll	OCI.DLL

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\DBExpress\dbxdrivers.ini.

6.3. Соединение с сервером баз данных

Для создания соединения с сервером в рамках технологии dbExpress приложение должно использовать 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 раскрывающегося списка конкретное настроенное соединение. По умолчанию разработчику доступно по одному настроенному соединению для каждого сервера БД. После выбора соединения автоматически устанавливаются значения следующих свойств (см. табл. 6.1):

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;

свойство

```
property Params: TStrings;
```

инкапсулирует параметры соединения, в его списке содержатся настройки для выбранного соединения.

При необходимости все перечисленные свойства можно установить самостоятельно.

Разработчик может дополнять и изменять список настроенных соединений. Для этого используется специализированный Редактор Соединений **dbExpress Coonections** (рис. 6.2). 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 на 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\DBExpress\dbxconnections.ini.

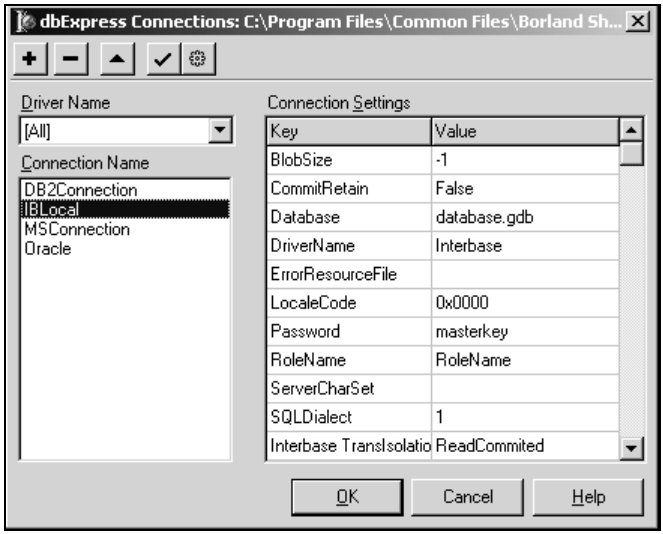


Рис. 6.2. Окно редактора соединений компонента `TSQLConnection`

Конкретные значения настроек соединений зависят от используемого сервера БД и требований приложения (табл. 6.2).

Таблица 6.2. *Настройка соединения dbExpress*

Параметр	Значение
	Общие настройки
`BlobSize`	Задает ограничение на объем пакета данных для данных типа BLOB
`DriverName`	Имя драйвера
`ErrorResourceFile`	Файл сообщений об ошибках
`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
	InterBase
`CommitRetain`	Задает поведение курсора после завершения транзакции. При значении `True` курсор обновляется, иначе — удаляется

Таблица 6.2 *(окончание)*

Параметр	Значение
`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 IntreBase 5 возможно только одно значение — 1
`WaitOnLocks`	Разрешение на ожидание занятых ресурсов
	MySQL
`Database`	Имя базы данных
`HostName`	Имя компьютера, на котором работает сервер MySQL
	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

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 диалоговое окно авторизации пользователя перед открытием соединения. При значении `True` окно 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. 6.2). 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.

```
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;
```

6.4. 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 хранится в свойстве

```
property MaxStmtsPerConn: LongWord;
```

Поэтому перед открытием набора данных можно выполнить 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 с помощью данного соединения 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 список параметров `TParams` и заполнить его. При этом, поскольку 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 помещает указатель на него в параметр `ResultSet`. Функция возвращает число обработанных запросом записей. Следующий фрагмент кода иллюстрирует 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.

Метод

```
procedure GetTableNames(List: TStrings; 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, используя метод

```
procedure GetFieldNames(const TableName: String; List: TStrings);
```

и список индексов при помощи метода

```
procedure GetIndexNames(const TableName: string; List: TStrings);
```

В обоих методах список возвращаемых значений содержится в параметре `List`.

Аналогичным образом метод

```
procedure GetProcedureNames(List: TStrings);
```

возвращает список доступных хранимых процедур, а метод

```
procedure GetProcedureParams(ProcedureName: String; List: TList);
```

определяет параметры отдельной процедуры.

6.5. Транзакции

Подобно своим аналогам в BDE и ADO компонент `TSQLConnection` поддерживает механизм транзакций и делает это сходным образом.

Запуск, 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` используется при работе с сервером Oracle.

Некоторые серверы БД не поддерживают транзакции, и для проверки этого факта используется свойство

```
property TransactionsSupported: LongBool;
```

Если соединение уже находится в транзакции, свойству

```
property InTransaction: Boolean;
```

присваивается значение `True`. Всегда полезно убедиться, что соединение не обслуживает начатую транзакцию, а особенно если сервер не поддерживает множественные транзакции:

```
var TransInfo: TTransactionDesc;
{...}
if Not MyConnection.InTransaction then
 try
  MyConnection.StartTransaction(TransInfo);
  {...}
  MyConnection.Commit(TransInfo);
 except
  MyConnection.Rollback(TransInfo);
 end;
```

6.6. Использование компонентов наборов данных

Набор компонентов dbExpress, содержащих набор данных, вполне обычен и сравним с аналогичными компонентами BDE, ADO, InterBase Express. Это компоненты `TSQLDataSet`, `TSQLTable`, `TSQLQuery`, `TSQLStoredProc`.

Примечание

Компонент `TSQLClientDataSet` также относится к рассматриваемой группе, но так как он обладает рядом специфических возможностей, его описание вынесено в отдельный раздел.

Однако необходимость создания легкой технологии доступа к данным, какой является dbExpress, наложила на эти компоненты ряд ограничений.

Несмотря на то, что общим предком всех перечисленных компонентов является класс `TDataSet`, который обладает полным инструментарием для работы с набором данных, компоненты dbExpress используют только одно-

направленные курсоры и не позволяют редактировать данные. Однонаправленные курсоры ограничивают навигацию по набору данных и обеспечивают перемещение только на следующую запись и возврат на первую. Для компонентов dbExpress также недоступны любые операции, требующие буферизации данных — это поиск, фильтрация, синхронный просмотр.

Таким образом, для выключения ряда механизмов класса `TDataSet` понадобился еще один промежуточный класс `TCustomSQLDataSet`.

Отображение данных при помощи компонентов со страницы **Data Controls** также ограничено. Нельзя использовать компоненты `TDBGrid` и `TDBCtrlGrid`, а в `TDBNavigator` не забудьте отключить кнопки возврата на одну позицию назад и перехода на последнюю запись. Ничего хорошего не выйдет из попытки применить компоненты синхронного просмотра. Остальные компоненты можно использовать обычным способом.

Доступ к данным во всех рассматриваемых компонентах осуществляется одинаково — через соединение, инкапсулированное в компоненте `TSQLConnection`. Привязку к соединению выполняет свойство

```
property SQLConnection: TSQLConnection;
```

Рассмотрим теперь компоненты dbExpress подробнее.

6.6.1. Класс *TCustomSQLDataSet*

Поскольку общим предком компонентов dbExpress является класс `TDataSet`, то задачей класса `TCustomSQLDataSet` является не столько внесение новой функциональности, сколько корректное ограничение возможностей, заложенных в `TDataSet`. Непосредственно в приложениях это класс не используется, но информация о нем полезна для понимания других компонентов dbExpress и для создания собственных компонентов на его основе.

Класс `TCustomSQLDataSet` является общим предком для компонентов, содержащих запросы, таблицы и хранимые процедуры. Для их поддержки используются свойства

```
TSQLCommandType = (ctQuery, ctTable, ctStoredProc);
property CommandType: TSQLCommandType;
```

определяющее 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 извлечение 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 использования набора данных в табличном представлении существует свойство

```
property SortFieldNames: string;
```

определяющее порядок сортировки записей табличного набора данных. Свойство должно содержать список полей, разделенных точкой с запятой. Это свойство используется для создания выражения `ORDER BY` в генерируемой команде.

Для обработки исключительных ситуаций в классах-потомках может применяться защищенное свойство

```
property LastError: string;
```

которое возвращает текст последней ошибки dbExpress.

Для ускорения работы набора данных можно отключить получение от сервера метаданных (информации о структуре данных) объекта запроса (таблицы, процедуры, полей, индексов), которые обычно направляются клиенту вместе с результатом запроса. Для этого свойству

```
property NoMetadata: Boolean;
```

присваивается значение `True`. Однако пользоваться им нужно осторожно, так как для некоторых видов команд метаданные необходимы (это операции с использованием индексов).

Разработчик может управлять процессом получения метаданных. Для этого необходимо заполнить структуру

```
TSchemaType = (stNoSchema, stTables, stSysTables, stProcedures, stCol-
umns, stProcedureParams, stIndexes);

  TSchemaInfo = record
    FType      : TSchemaType;
    ObjectName : String;
    Pattern    : String;
  end;
```

которая содержится в защищенном свойстве

```
property SchemaInfo: TSQLSchemaInfo;
```

а следовательно может использоваться 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 компоненту нужно получить результирующий набор данных, параметр FType должен обязательно иметь значение stNoSchema. При изменении значения свойства CommandText это требование 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 _.

Если нужно использовать управляющие символы в качестве маскирующих, то 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.

6.6.2. Компонент *TSQLDataSet*

Компонент TSQLDataSet является универсальным и позволяет выполнять запросы SQL (подобно TSQLQuery), просматривать таблицы целиком (подобно TSQLTable) или выполнять хранимые процедуры (подобно TSQLStoredProc).

Для определения режима работы компонента используется уже описанное свойство CommandType.

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 используется значение ctStoredProc, а в списке свойства CommandText следует выбрать нужную процедуру.

Для открытия набора данных применяются традиционные способы: свойство Active или метод Open. Если же запрос SQL или хранимая процедура не возвращают набор данных, для их выполнения используется 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 компонента можно использовать свойство SortFieldNames, определяющее порядок сортировки записей таблицы.

В режиме запросов и хранимых процедур для задания параметров используются свойства `Params` и `ParamCheck`.

Информация о применяемых в результирующем наборе данных индексах сохраняется в свойстве

```
property IndexDefs: TIndexDefs;
```

Подробную информацию о типе `TIndexDefs` см. в гл. 5.

6.6.3. Компонент *TSQLTable*

Компонент `TSQLTable` предназначен для просмотра таблиц целиком и по основным функциям подобен своим аналогам `TTable`, `TADOTable`, `TIBTable`. *Подробно работа с табличными компонентами рассмотрена в гл. 5.*

Для определения имени таблицы используется свойство

```
property TableName: string;
```

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 (свойство `Active` или метод `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 используются 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 используется метод

```
procedure DeleteRecords;
```

6.6.4. Компонент *TSQLQuery*

Компонент `TSQLQuery` позволяет выполнять на сервере запросы SQL клиента. Все его возможности совпадают со стандартными для компонентов данного типа *(см. гл. 5)*.

Настройка соединения 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;
```

Для редактирования текста запроса используется стандартный редактор, открывающийся при щелчке на кнопке свойства в Инспекторе объектов.

Параметры запроса должны содержаться в индексированном свойстве

```
property Params: TParams;
```

Для подготовки параметров и запроса в целом на сервере используется свойство

```
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 создаваться заново список параметров запроса, если текст запроса был изменен. Например, запрос применяется для создания хранимых процедур, исходный текст которых передается на сервер в запросе. В теле хранимой процедуры могут использоваться собственные параметры, которые при анализе запроса могут быть интерпретированы как параметры самого запроса. В этом случае, конечно, возникнет ошибка создания параметров. В такой ситуации очень уместно использовать 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 используется метод

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Параметр `ExecDirect = False` означает, что запрос не имеет настраиваемых параметров.

6.6.5. Компонент *TSQLStoredProc*

Компонент `TSQLStoredProc` инкапсулирует функциональность хранимых процедур для их выполнения в рамках технологии dbExpress. Все его функции стандартны. *Подробную информацию о функциях компонентов хранимых процедур см. в гл. 5.*

Настройка соединения 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 в Инспекторе объектов из раскрывающегося списка свойства.

Для работы с входными и выходными параметрами предназначено свойство

```
property Params: TParams;
```

Внимание!

При работе с параметрами желательно использовать обращение 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 создаваться заново 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 включения новых данных. Возврат к предыдущему набору данных возможен, если вы определили объектные переменные для каждого набора данных:

```
var SecondSet: TCustomSQLDataSet;
...
 MyProc.Open;
 while Not MyProc.Eof do
 begin
  {...}
  Next;
 end;
 SecondSet := MyProc.NextRecordSet;
 SecondSet.Open;
 {...}
 SecondSet.Close;
 MyProc.Close;
```

6.7. Компонент *TSQLClientDataSet*

Компонент `TSQLClientDataSet` 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 *(см. гл. 10)*, компонент `TSQLClientDataSet` призван быть лишь средством редактирования набора данных в технологии dbExpress.

В отличие от других компонентов dbExpress, компонент `TSQLClientDataSet` использует двунаправленный курсор и позволяет редактировать данные, правда только в режиме кэширования. Набор данных помещается в локальный буфер компонента, и все текущие изменения сохраняются в нем. Если нужно сохранить изменения на сервере, то используется специальный метод, пересылающий изменения серверу.

Таким образом, компонент `TSQLClientDataSet` позволяет исправить основные недостатки технологии dbExpress.

6.7.1. Соединение с сервером БД

Для подключения к источнику данных компонент `TSQLClientDataSet` использует свойство

```
property DBConnection: TSQLConnection;
```

которое позволяет связать его с соединением `TSQLConnection`. Или свойство

```
property ConnectionName: string;
```

которое дает возможность выбрать тип соединения dbExpress напрямую.

При этом у компонента `TSQLClientDataSet` отсутствует механизм создания удаленного доступа к данным, представленный в компоненте `TClientDataSet` свойствами `RemoteServer` и `ProviderName` *(см. гл. 10)*.

После создания соединения с сервером БД можно определить тип используемой команды, подобно тому, как это делалось для компонента `TSQLDataSet`.

Тип команды определяется свойством

```
TSQLCommandType = (ctQuery, ctTable, ctStoredProc);
property CommandType: TSQLCommandType;
```

А содержание команды задает свойство

```
property CommandText: string;
```

После этого компонент `TSQLClientDataSet` можно связывать с компонентами отображения информации и просматривать и редактировать данные.

6.7.2. Сохранение изменений

Для передачи на сервер сделанных и сохраненных в локальном кэше изменений используется метод

```
function ApplyUpdates(MaxErrors: Integer); Integer; virtual;
```

где параметр `MaxErrors` определяет максимально возможное число ошибок при сохранении. Если число возникших ошибок сохранения превышает число `MaxErrors`, процесс сохранения прерывается. 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. Обычно использовать этот метод нет необходимости, так как он вызывается методом `ApplyUpdates`.

До и после сохранения изменений на сервере вызываются методы-обработчики

```
type
  TRemoteEvent = procedure(Sender: TObject; var OwnerData: OleVariant) of object;
```

```
property BeforeApplyUpdates: TRemoteEvent;
property AfterApplyUpdates: TRemoteEvent;
```

Отменить локальные изменения можно методом

```
procedure CancelUpdates;
```

Обратите внимание на то, что в компоненте `TSQLClientDataSet` действуют традиционные методы набора данных `Edit`, `Post`, `Cancel`, `Apply`, `Insert`, `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 `TSQLClientDataSet` 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 этот размер. Размер пакета определяется свойством

```
property PacketRecords: Integer;
```

которое задает число записей в пакете. Автоматическое назначение размера 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 используется метод

```
function GetNextPacket: Integer;
```

До и после использования этого метода вызываются методы-обработчики событий

```
property BeforeGetRecords: TRemoteEvent;
property AfterGetRecords: TRemoteEvent;
```

Для организации такой подкачки можно, например, использовать метод-обработчик `AfterScroll`:

```
procedure TDM.SQLClientDataSetAfterScroll(DataSet: TDataSet);
begin
 if SQLClientDataSet.Eof
  then SQLClientDataSet.GetNextPacket;
end;
```

6.7.3. Работа с записями

В компоненте TSQLClientDataSet развиты средства работы с одиночными записями. Во-первых, 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 с сервера значения полей для текущей записи клиента можно методом.

```
procedure RefreshRecord;
```

До и после метода RefreshRecord вызываются методы-обработчики

```
property BeforeRowRequest: TRemoteEvent;
property AfterRowRequest: TRemoteEvent;
```

6.7.4. Обработка исключительных ситуаций

Обработка исключительных ситуаций для компонента TSQLClientDataSet состоит из двух этапов.

Во-первых, необходимо отслеживать ошибки на стороне клиента — это могут быть некорректный ввод данных, ошибки кэширования и т. д. В этом случае подходят все стандартные способы, применяемые для наборов данных. Вы можете использовать блоки try..except:

```
...
try
  DM.SQLClientDataSet.Edit;
  DM.SQLClientDataSet.Fields[1].AsString := SomeString;
  DM.SQLClientDataSet.Post;
except
  on E: EDatabaseError do DM.SQLClientDataSet.Cancel;
 end;
...
```

Во-вторых, ошибки могут возникнуть при сохранении изменений на сервере. И так как само событие, приведшее к исключительной ситуации, возникает на другом компьютере или в другом процессе, для отслеживания таких ошибок используется 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`. Например:

```
procedure TDM.SQLClientDataSetReconcileError(DataSet: TCustomClientData-
Set; E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcil-
eAction);
begin
 if (E.ErrorCode = SomeCode)and(UpdateKind = ukModify) then
 begin
  MessageDlg('Server Error', mtError, [mbOK], 0);
  Action := raCorrect;
 end
 else
 begin
  Action := raCancel;
 end;
end;
```

Более подробная информация о клиентских наборах данных содержится в гл. 10.

6.8. Способы редактирования данных

Несмотря на декларированные недостатки технологии dbExpress — однонаправленные курсоры и невозможность редактирования, существуют программные способы решить проблему частично или полностью.

Во-первых, в нашем распоряжении есть компонент `TSQLClientDataSet`, 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 `TSQLClientDataSet` безусловно хорош. Он технологичен, относительно прост в использовании и, главное, прячет всю функциональность в нескольких свойствах и методах. Но локальное кэширование изменений подходит далеко не для всех приложений.

Например, при интенсивном многопользовательском доступе к данным при их редактировании с помощью локального кэширования 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. 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.

Использование 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 в этом случае существенно выше.

Рассмотрим небольшой пример реализации обоих способов (листинг 6.1). Приложение Demo_DBX использует соединение с сервером InterBase. Подключена тестовая база данных ..\Program Files\Common Files\Borland Shared\Data\MastSQL.gdb.

Листинг 6.1. Пример 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 (Поставщики) и CUSTOMERS (Получатели) (рис. 6.3). Первая таблица подключена через настроенное соединение (компонент `cnMast`) к компоненту `tblVens` типа `TSQLTable`. Значения пяти полей отображаются в обычных компонентах `TEdit`, так как компоненты отображения данных, связанные с компонентом dbExpress через компонент `TDataSource`, работают только в режиме просмотра, не позволяя редактировать данные.

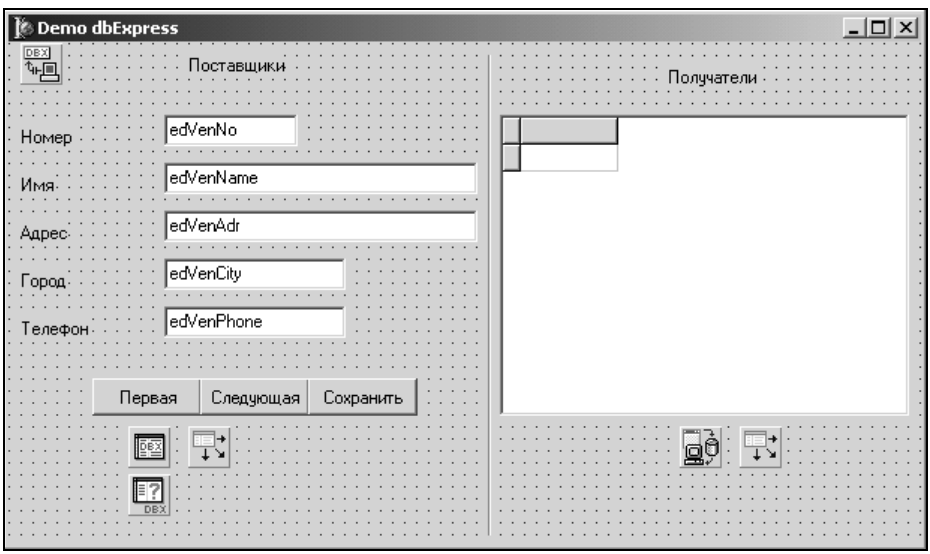


Рис. 6.3. Окно приложения Demo_DBX

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`.

Вторая таблица подключена через тот же компонент `cnMast` к компоненту `cdsCusts` типа `TSQLClientDataSet`. Он работает в табличном режиме. Данные отображаются в обычном компоненте `TDBGrid`.

Для сохранения сделанных изменений после того, как они уже попали в локальный кэш, здесь использован метод `ApplyUpdates`, размещенный в методе-обработчике `AfterPost`. Метод-обработчик вызывается каждый раз при переходе в компоненте `TDBGrid` на новую строку.

Для компонента `cdsCusts` предусмотрена простейшая обработка исключительных ситуаций, возникающих на сервере.

В Инспекторе объектов для компонента `cnMast` типа `TSQLConnection` надо установить свойства `KeepConnection` и `LoginPrompt` со значениями `False`, что обеспечивает с минимальным исходным кодом открытие наборов данных при создании формы и автоматическое закрытие соединения при закрытии приложения.

6.9. Интерфейсы dbExpress

Технология dbExpress основана на использовании четырех базовых интерфейсов, методы которых применяются во всех компонентах dbExpress. При разработке серьезных проектов или создании собственных компонентов информация об этих интерфейсах будет полезна.

6.9.1. Интерфейс *ISQLDriver*

Интерфейс `ISQLDriver` содержит всего три метода для обслуживания драйвера dbExpress. Экземпляр интерфейса создается для соединения и обеспечивает связь соединения 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 с помощью защищенного свойства

```
property Driver: ISQLDriver read FSQLDriver;
```

компонента `TSQLConnection`.

6.9.2. 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;
```

где `pszServerName` — имя базы данных, `pszUserName` и `pszPassword` — имя и пароль пользователя.

Закрывает соединение метод

```
function disconnect: SQLResult; stdcall;
```

Параметры соединения устанавливаются и определяются методами, соответственно

```
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;
```

При помощи метода

```
function getErrorMessage(Error: PChar): SQLResult; overload; stdcall;
```

организована обработка исключительных ситуаций в компоненте `TSQLConnection`. В интерфейсе реализована защищенная процедура `SQLError`, которую можно использовать в собственных компонентах и при необходимости дорабатывать.

Например, можно написать собственную процедуру контроля ошибок примерно по такому образцу:

```
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 с помощью свойства

```
property SQLConnection: ISQLConnection;
```

компонента `TSQLConnection`.

6.9.3. Интерфейс *ISQLCommand*

Интерфейс `ISQLCommand` обеспечивает функционирование запроса dbExpress. Компоненты dbExpress, работающие с наборами данных, используют его для реализации своих методов.

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;
```

где `uParameterNumber` — порядковый номер параметра; если параметр является дочерним для сложных типов данных, `uChildPos` задает его порядковый номер; `ePType` задает вид 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.

Метод вызывается для каждого параметра.

Информацию о параметре можно получить методом

```
function getParameter(ParameterNumber: Word; ulChildPos: Word; Value:
Pointer; Length: Integer; var IsBlank: Integer): SQLResult; stdcall;
```

где `uParameterNumber` — порядковый номер параметра; если параметр является дочерним для сложных типов данных, `uChildPos` задает его порядковый номер; `pData` — указатель на буфер значения параметра; `ulLength` 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, возвращающая несколько наборов данных.

Интерфейс `ISQLCommand` используется компонентом `TCustomSQLDataSet` и недоступен потомкам.

6.9.4. Интерфейс *ISQLCursor*

Интерфейс `ISQLCursor` обладает совокупностью методов, которые помогут получить информацию о полях курсора набора данных, а также значения этих полей. Все эти методы имеют одинаковое представление. Для получения нужной информации следует 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.

6.10. Отладка приложений с технологией dbExpress

Наряду с обычными методами отладки исходного кода, в dbExpress существует возможность контроля запросов, передаваемых на сервер через соединение. Для этого используется компонент `TSQLMonitor`.

С помощью свойства

```
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 после записи в список вызывается метод

```
TTraceLogEvent = procedure (Sender: TObject; CBInfo: pSQLTRACEDesc) of object;
property OnLogTrace: TTraceLogEvent;
```

Таким образом, разработчик получает компактный и симпатичный 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.

Метод 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.

```
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 использует вызовы метода `SetTraceCallbackEvent`, поэтому одновременно использовать компонент и собственные функции нельзя.

6.11. 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. столбец "Драйвер" табл. 6.1)*. Она находится в папке ..\Delphi6\Bin.

Дополнительно, если в приложении используется компонент `TSQLClientDataSet`, необходимо включить в поставку динамическую библиотеку midas.dll.

Приложение компилируется вместе со следующими DCU-файлами: dbExpInt.dcu, dbExpOra.dcu, dbExpDb2.dcu, dbExpMy.dcu (в зависимости от выбранного сервера). Если в приложении используется компонент `TSQLClientDataSet`, 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 четырех серверов баз данных.

- DB2
- InterBase
- MySQL
- Oracle

Технология dbExpress использует стандартные типы компонентов доступа к данным, и приложения, применяющие ее, легко распространяются (исполняемый файл приложения или одна-две динамические библиотеки). Поддерживает кросс-платформенную разработку для Linux и легко интегрируется в приложения CLX.

К недостаткам технологии нужно отнести использование однонаправленных курсоров и ограниченные возможности для редактирования (редактирование возможно только при кэшировании изменений на стороне клиента или выполнении специальных модифицирующих запросов).

Глава 7

Использование ADO средствами Delphi

Наряду с традиционными инструментами доступа к данным Borland Database Engine и ODBC (Open Database Connectivity) в приложениях Delphi можно применять технологию *Microsoft ActiveX Data Objects (ADO)*, которая основана на возможностях COM, а именно интерфейсов OLE DB.

Технология ADO завоевала популярность у разработчиков благодаря универсальности — базовый набор интерфейсов OLE DB есть в каждой современной операционной системе фирмы 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 Палитре компонентов Delphi есть страница ADO, содержащая набор компонентов, позволяющих создавать полноценные приложения БД, обращающиеся к данным через ADO.

В этой главе рассматриваются следующие вопросы.

- Что такое технология ADO, каков перечень доступных провайдеров ADO, а также работающих в ней объектов и интерфейсов?
- Как создать соединение с базой данных с помощью ADO в приложении Delphi?
- Как применять объект набора записей ADO в приложении?
- Как использовать таблицы, запросы SQL и хранимые процедуры?
- Что такое команды и объекты команды ADO?

7.1. Основы ADO

Технология Microsoft ActiveX Data Objects обеспечивает универсальный доступ к источникам данных из приложений БД. Такую возможность предоставляют функции совокупности интерфейсов, созданной на основе общей модели объектов COM и описанной в спецификации OLE DB.

Технология ADO и интерфейсы OLE DB обеспечивают для приложений единый способ доступа к источникам данных различных типов (рис. 7.1). Например, приложение, использующее ADO, может применять одинаково сложные операции и к данным, хранящимся на корпоративном сервере SQL, и к электронным таблицам и локальным СУБД. Запрос SQL, направленный через ADO любому источнику данных, будет выполнен.

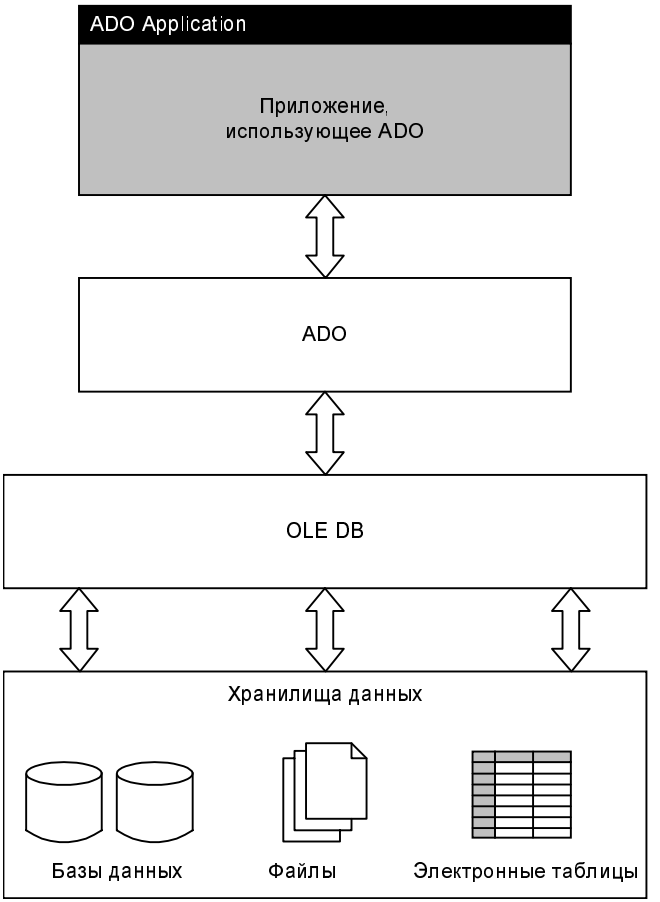


Рис. 7.1. Схема доступа к данным через ADO

Возникает вопрос: каким образом источники данных смогут выполнить этот запрос?

О серверах БД беспокоиться 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, включающих в себя стандартные функции обработки данных и специализированные функции конкретных источников данных, и интерфейсов, обеспечивающих передачу данных между объектами.

Согласно терминологии ADO любой источник данных (база данных, электронная таблица, файл) называется хранилищем данных, с которым при помощи 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 используется метод `CoCreateInstance` и соответствующая фабрика класса. Объекту соответствует совокупность интерфейсов, к методам которых можно обращаться после создания объекта. *Подробную информацию о работе с объектами COM см. в гл. 1.*

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. Необходимые объекты и соответствующий провайдер могут быть созданы для любого хранилища данных без внесения изменений в исходную структуру ADO. При этом существенно расширяется само понятие данных — ведь можно разработать совокупность 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.

Поскольку технология ADO основана на стандартных интерфейсах COM, которые являются системным механизмом Windows, общий объем работающего программного кода сокращается и приложения БД можно распространять 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.

- *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)* включает в себя 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 ошибки (Error)* содержит информацию об исключительной ситуации.

Рассмотрим функциональные возможности основных объектов и интерфейсов OLE DB.

7.1.1. Перечислители

Объекты-перечислители выполняют поиск любых объектов ADO, которые обеспечивают доступ к источникам данных. При этом другие перечислители также доступны в данном перечислителе.

Первичный поиск источников данных осуществляется в провайдере ADO. Перечислители могут отбирать только источники данных конкретных типов, поскольку провайдер обеспечивает доступ к конкретному типу хранилища данных.

В составе ADO есть *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 ...\Delphi6\Source\Vcl\OleDB.pas. Его описание выглядит следующим образом:

```
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.

7.1.2. 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 предоставляет информацию о требуемом реальном источнике данных и выполняет подключение к нему.

Для ввода сведений о хранилище данных используется интерфейс `IDBProperties`. Вероятно, для любого хранилища данных обязательна информация о его имени, пользователе и пароле. Кроме того, 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`.

Инициализируется соединение методом

```
function Initialize: HResult; stdcall;
```

интерфейса `IDBInitialize` объекта-источника данных.

7.1.3. Сессия

Объект-источник данных позволяет создавать объекты-сессии. Для этого используется метод интерфейса `IDBCreateSession`

```
function CreateSession(const punkOuter: IUnknown; const riid: TGUID; out
ppDBSession: IUnknown): HResult; stdcall;
```

Сессия предназначена для обеспечения работы транзакций и наборов рядов.

7.1.4. Транзакции

Управление транзакциями в OLE DB реализовано на двух уровнях.

Во-первых, всеми необходимыми методами обладает объект-сессия. Он имеет интерфейсы `ITransaction`, `ITransactionJoin`, `ITransactionLocal`, `ITransactionObject`.

Внутри сессии транзакция управляется интерфейсами `ITransactionLocal`, `ItransactionSC`, `ITransaction` и их методами `StartTransaction`, `Commit`, `Rollback`.

Во-вторых, для объекта-сессии можно создать объект-транзакцию при помощи метода интерфейса `ITransactionObject`

```
function GetTransactionObject(ulTransactionLevel: UINT;
out ppTransactionObject: ITransaction): HResult; stdcall;
```

который возвращает ссылку на интерфейс объекта-транзакции.

7.1.5. Наборы рядов

Объект-набор рядов является основным объектом ADO, обеспечивающим работу с данными. Он содержит совокупность рядов из источника данных и механизмы навигации по рядам и поддержания рядов в актуальном состоянии (согласование данных клиента с сервером).

Объект-сессия 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` предоставляет информацию о формате рядов, содержащихся в буфере набора рядов;
- `IRowsetInfo` позволяет получить сведения о наборах рядов (например, число рядов или число обновленных рядов);
- `IColumnsInfo` предоставляет 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 для управления набором рядов предоставляют следующие интерфейсы:

- `IRowsetChange` модифицирует набор рядов (вносит изменения, добавляет новые ряды, удаляет ряды и т. д.);
- `IRowsetIdentity` позволяет сравнивать ряды разных наборов рядов;
- `IRowsetIndex` обеспечивает использование индексов;
- `IRowsetLocate` выполняет поиск в наборе рядов;
- `IRowsetUpdate` реализует механизм кэширования изменений.

7.1.6. Команды

Программные средства ADO были бы неполными, если бы не могли использовать для работы с данными язык SQL. Операторы *DML (Data Manipulation Language, язык управления данными)* и *DDL (Data Definition Language, язык определения данных)*, ряд специальных операторов ADO носят общее название *текстовых команд*.

Объект-команда содержит саму текстовую команду и механизм обработки и передачи команды. Объект-команда выполняет следующие операции:

- разбор текста (выделение отдельных операторов, параметров и т. д.) команды;
- связывание команды с источником данных;
- оптимизацию команды;
- передачу 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-команда 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` содержит методы для получения и установки текста команды (этот интерфейс обязателен для объекта-команды).
- `ICommandWithParameters` обеспечивает работу с параметрами команды.

7.2. Провайдеры ADO

Провайдеры ADO выполняют с помощью ADO соединение приложения, использующего данные, с хранилищем (сервером SQL, локальной СУБД, файловой системой и т. д.). Для каждого типа источника 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 для передачи их приложению.

Список установленных в данной операционной системе провайдеров предоставляется при установке соединения в компоненте `TADOConnection`.

При инсталляции Microsoft ActiveX Data Objects в операционной системе устанавливаются следующие стандартные провайдеры.

- *Microsoft Jet OLE DB Provider* — обеспечивает соединение с данными СУБД Access с помощью технологии DAO.
- *Microsoft OLE DB Provider for Microsoft Indexing Service* — делает доступными только для чтения файлы и Internet-ресурсы 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 использовать HTTP-файлы и ресурсы, предоставляемые Microsoft FrontPage, Microsoft Internet Information Server.
- *Microsoft Data Shaping Service for OLE DB* — предоставляет возможность 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* — позволяет работать с данными, которые уже доступны благодаря применению 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* — обеспечивает соединение с сервером Oracle.
- *Microsoft OLE DB Provider for SQL Server* — предназначен для соединения с сервером MIcrosoft SQL Server.

7.3. Реализация ADO в Delphi

Механизм доступа к данным с помощью ADO и многочисленные объекты и интерфейсы реализованы в VCL Delphi в виде набора компонентов, расположенных на странице **ADO**. Все необходимые интерфейсы, обеспечивающие работу компонентов, объявлены и описаны в файлах OleDB.pas и ADODB.pas в папке ..\Delphi6\Source\Vcl.

7.3.1. Компоненты ADO

Компонент `TADOConnection` вобрал в себя возможности перечислителя, источника данных и сессии, способной обслуживать транзакции.

Текстовые команды ADO реализованы в компоненте `TADOCommand`.

Наборы рядов (нотация Microsoft) можно получить при помощи компонентов `TADOTable`, `TADOQuery`, `TADOStoredProc`. Каждый из них реализует способ доступа к конкретному типу представления данных в хранилище. Применительно к компонентам Delphi совокупность строк, возвращаемых из хранилища данных, называется набором записей, что соответствует документации Borland *(см. **www.borland.com** или **www.borland.ru**)*. Поскольку с этого момента мы будем обсуждать реализацию ADO в Delphi, целесообразно использовать традиционный для Delphi термин "набор данных" вместо "набора записей", стандартного для ADO.

Совокупность свойств и методов компонентов ADO обеспечивает реализацию всех необходимых приложению БД функций. Способы использования компонентов ADO немногим отличаются от применения стандартных VCL-компонентов доступа к данным *(см. гл. 5)*.

Однако при необходимости разработчик может использовать все возможности интерфейсов ADO, используя соответствующие объекты ADO. Ссылки на объекты есть в компонентах.

7.3.2. Механизм соединения с хранилищем данных ADO

Компоненты доступа к данным ADO могут применять два варианта подключения к хранилищу данных: 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, обязательно указание имен провайдера соединения или удаленного сервера:

```
ConnectionString:='Remote Server=ServerName;Provider=ProviderName';
```

При необходимости задаются 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, задает параметры соединения в свойстве `ConnectionString` и открывает собственное соединение. Чем больше приложение содержит компонентов ADO, тем больше соединений может быть открыто одновременно.

Поэтому целесообразно реализовать механизм соединения ADO с помощью специального компонента — `TADOConnection`. Этот компонент открывает соединение, также 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 есть у каждого компонента, содержащего набор данных ADO.

7.4. Компонент *TADOConnection*

Компонент `TADOConnection` предназначен для управления соединением с объектами хранилища данных ADO. Он обеспечивает доступ к хранилищу данных компонентам ADO, содержащим набор данных.

Применение этого компонента дает разработчику ряд преимуществ:

- обращение всех компонентов доступа к данным ADO к хранилищу данных через одно соединение;

- возможность напрямую задать объект провайдера соединения;
- доступ к объекту соединения ADO;
- возможность выполнять команды ADO;
- управление транзакциями;
- расширенное управление соединением при помощи методов-обработчиков событий.

7.4.1. Настройка соединения

Перед открытием соединения необходимо задать его параметры. Для этого предназначено свойство

```
property ConnectionString: WideString;
```

которое подробно рассматривалось в предыдущем разделе. Добавим лишь, что набор параметров изменяется в зависимости от типа провайдера и может настраиваться как вручную, так и при помощи специального Редактора настройки соединения (рис. 7.2), который вызывается двойным щелчком на компоненте `TADOConnection`, перенесенным на форму, или щелчком на кнопке в поле редактирования свойства `ConnectionString` в Инспекторе объектов.

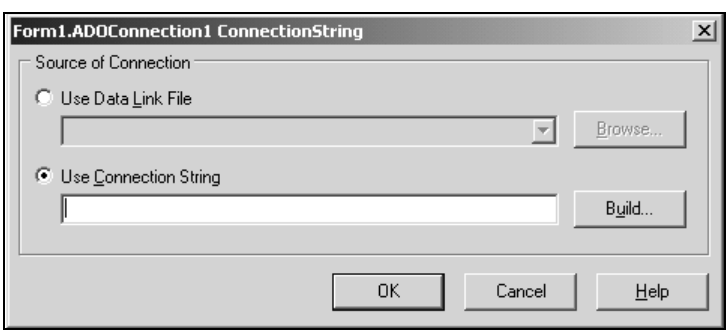


Рис. 7.2. Редактор настройки соединения ADO

Здесь можно настроить соединение, используя свойство `ConnectionString` (радиокнопка **Use Connection String**), или загрузить параметры соединения из файла с расширением UDL (радиокнопка **Use Data Link File**).

Файл UDL (листинг 7.1) представляет собой обычный текстовый файл, в котором указывается название параметра и через знак равенства его значение. Параметры разделяются точкой с запятой.

Листинг 7.1. 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 осуществлять вручную. Для этого следует нажать кнопку **Build**. В результате появляется диалоговое окно **Data Link Properties** (рис. 7.3), в котором можно настроить параметры соединения. Оно представляет собой четырехстраничный блокнот, позволяющий вам последовательно задать все необходимые параметры.

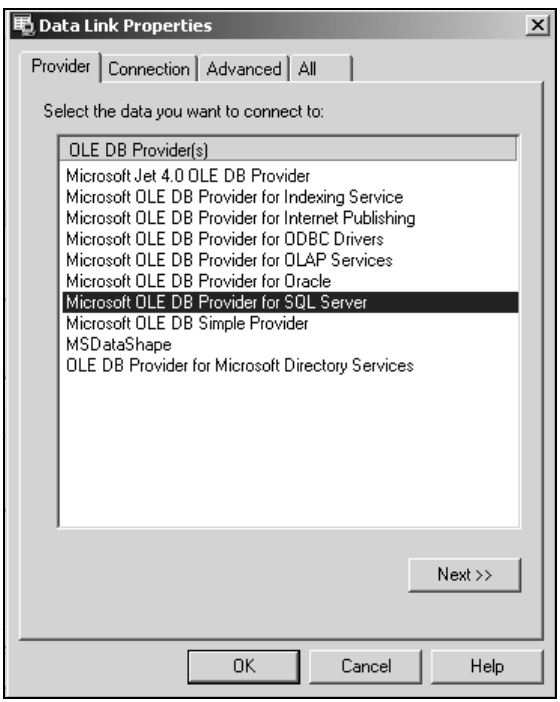


Рис. 7.3. Диалоговое окно настройки параметров соединения на странице выбора провайдера

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 для 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 аутентификации пользователя, а также определить имя и пароль пользователя.

Следующая страница **Connection** (рис. 7.4) настраивает источник данных. Рассмотрим процесс настройки на примере провайдера OLE DB для сервера Microsoft SQL Server.

1. Выберите имя сервера из списка доступных для данного компьютера.
2. Определите режим аутентификации пользователя. Это либо система безопасности Windows, либо собственная система аутентификации сервера. Здесь же надо определить имя и пароль пользователя.
3. Задайте базу данных сервера.

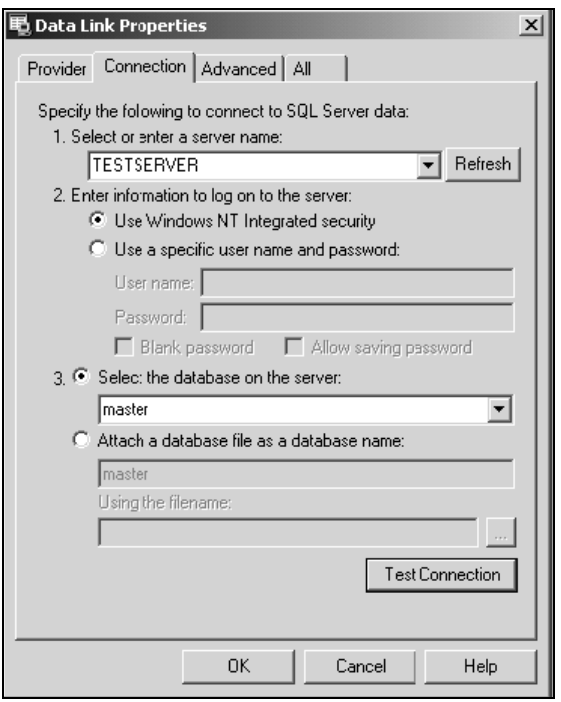


Рис. 7.4. Диалоговое окно настройки параметров соединения на странице выбора источника данных

После окончания настройки источника данных вы можете проверить соединение, нажав кнопку **Test Connection**.

Теперь перейдем на следующую страницу.

Страница **Advanced** (рис. 7.5) задает дополнительные параметры соединения. В зависимости от типа хранилища данных некоторые элементы этой страницы могут быть недоступны.

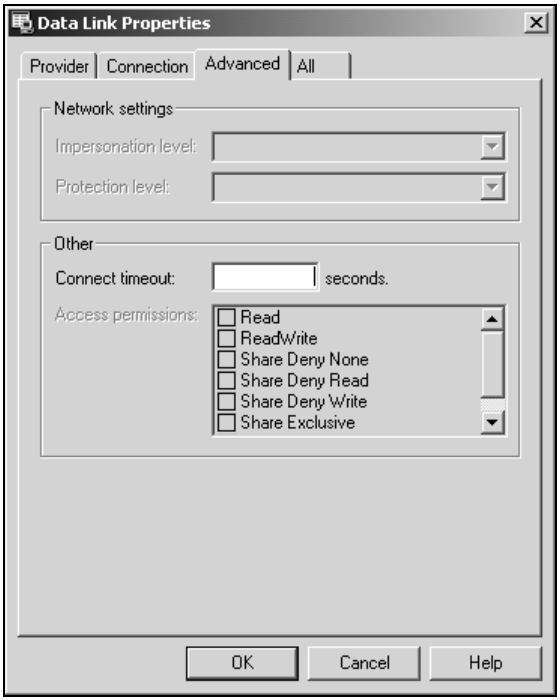


Рис. 7.5. Диалоговое окно настройки параметров соединения на странице дополнительных параметров

Список **Impersonation Level** определяет возможности клиентов при подключении в соответствии с полномочиями их ролей. В списке 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. В списке можно выбрать 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 запрещены всем;
- **Write** — только запись.

Последняя страница **All** (рис. 7.6) позволяет просмотреть и при необходимости изменить все сделанные настройки (для этого предназначена кнопка **Edit Value...**) для выбранного провайдера.

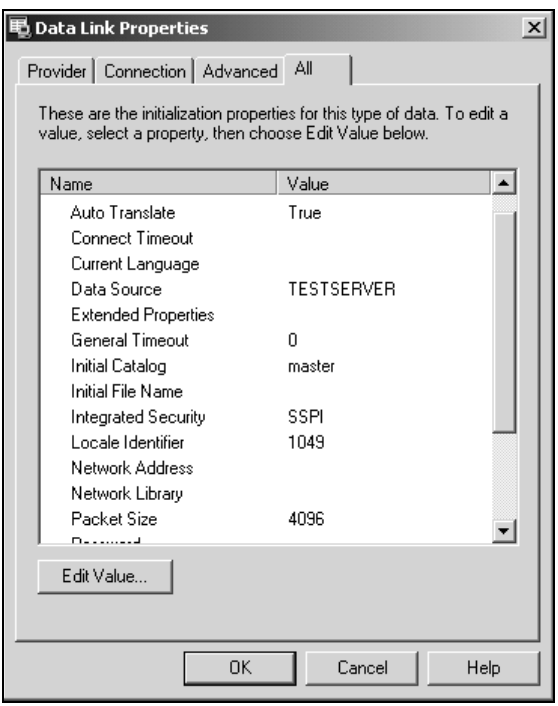


Рис. 7.6. Диалоговое окно настройки параметров соединения на странице просмотра настроек

После подтверждения выполненных в диалоговых окнах настроек из них формируется значение свойства `ConnectionString`.

7.4.2. 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 и определяется видом соединения: удаленное или локальное соединение. Для удаленного соединения применяется вариант с параметрами `UserID` и `Password`.

До и после открытия и закрытия соединения разработчик может использовать соответствующие стандартные методы-обработчики событий:

```
property BeforeConnect: TNotifyEvent;
property BeforeDisconnect: TNotifyEvent;
property AfterConnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
```

Кроме уже описанных, компонент `TADOConnection` имеет дополнительные методы-обработчики. После получения от провайдера подтверждения о том, что соединение будет открыто, и перед его реальным открытием вызывается метод

```
TWillConnectEvent = procedure(Connection: TADOConnection; var Connection-
String, 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, это определяется значением параметра `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);
```

- `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, режим синхронности/асинхронности можно также задать при помощи свойства

```
ConnectOptions property ConnectOptions: TConnectOption;
```

После открытия соединения для выполнения собственного кода используется метод-обработчик

```
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error; var EventStatus: TEventStatus) of object;
property OnConnectComplete: TConnectErrorEvent;
```

Если в процессе открытия соединения возникла ошибка, параметр `EventStatus` будет равен `esErrorsOccured`, а в параметре `Error` находится объект ошибки ADO.

Теперь перейдем к вспомогательным свойствам и методам компонента `TADOConnection`, обеспечивающим соединение.

Ограничения времени открытия соединения для медленных каналов связи устанавливает свойство

```
property ConnectionTimeout: Integer;
```

задающее время ожидания открытия соединения в секундах. По умолчанию оно равно 15 секундам.

Также можно определить реакцию компонента на неиспользуемое соединение. Если через соединение не подключен ни один активный компонент, свойство

```
property KeepConnection: Boolean;
```

равное `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 этого свойства.

Прямой доступ к объекту соединения OLE DB можно задать в свойстве

```
property ConnectionObject: _Connection;
```

При открытии соединения необходимо вводить имя пользователя и его пароль. Появление стандартного диалогового окна для ввода этой информации управляется свойством

```
property LoginPrompt: Boolean;
```

Для задания этих параметров без открытия диалогового окна можно использовать свойство `ConnectionString`, метод `Open` или метод-обработчик

```
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.

7.4.3. Доступ к связанным наборам данных и командам ADO

Компонент `TADOConnection` обеспечивает доступ ко всем компонентам, которые используют его для связи с хранилищем данных ADO. Все открытые таким образом наборы данных указаны в индексированном свойстве

```
property DataSets[Index: Integer]: TCustomADODataSet;
```

Каждый элемент этого списка содержит дескриптор компонента доступа к данным ADO (тип `TCustomADODataSet`). Общее число компонентов, связанных с наборами данных возвращается свойством

```
property DataSetCount: Integer;
```

Для этих компонентов можно централизованно установить тип используемого курсора 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` определяет 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;
...
```

Компонент `TADOConnection` обеспечивает функционирование не только наборов записей, но и команд ADO. Команду ADO содержит специальный компонент `TADOCommand`, который рассматривается ниже. Все команды ADO, работающие с хранилищем данных через данное соединение, перечислены в индексированном свойстве

```
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, реализуя таким образом нечто вроде сценария (скрипта):

```
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;
```

Компонент `TADOConnection` может выполнять команды ADO и самостоятельно, без помощи других компонентов. Для этого используется один из вариантов перегружаемого метода

```
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 ждать 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 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 ExecuteOptions: TExecuteOptions; var EventStatus: TEventStatus;
const Command: _Command; const 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; const Command: _Command; const Recordset: _Recordset) of object;

property OnExecuteComplete: TExecuteCompleteEvent;
```

7.4.4. Объект ошибок ADO

Все ошибки времени выполнения, возникающие при открытом соединении, сохраняются в специальном объекте ADO, содержащем коллекцию сообщений об ошибках. Доступ к объекту возможен с помощью свойства

```
property Errors: Errors;
```

Более подробную информацию об объекте ошибок ADO см. в разд. 7.7 данной главы.

7.4.5. 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.

Наличие нескольких значений свойства `Attributes`, задающих один и тот же уровень изоляции, обусловлено проблемами совместимости при использовании ADO различными программными платформами.

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.

7.5. Наборы данных ADO

На странице **ADO** Палитры компонентов Delphi кроме компонентов соединения есть стандартные компоненты, инкапсулирующие набор данных и адаптированные для работы с хранилищем данных ADO.

- `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` (рис. 7.7), предоставляющий базовые функции управления набором данных *(см. гл. 5)*.

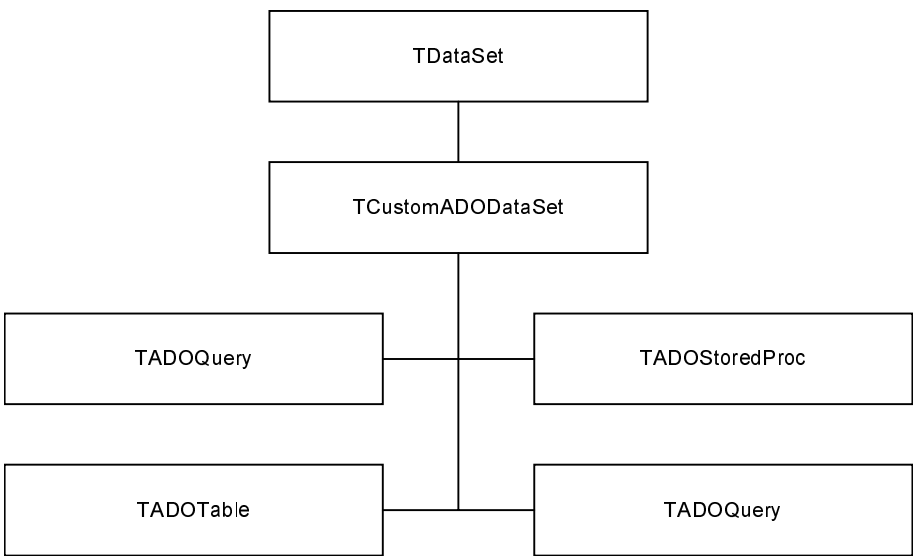


Рис. 7.7. Иерархия классов наборов данных ADO

Компоненты ADO обладают обычным набором свойств и методов, а механизм, необходимый для доступа к данным через ADO, 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. Сначала мы изучим класс `TCustomADODataSet` и только потом перейдем к компонентам ADO.

7.5.1. Класс *TCustomADODataSet*

Класс `TCustomADODataSet` содержит механизм доступа к хранилищу данных через ADO. Этот класс наполняет абстрактные методы общего предка `TDataSet` функциями конкретного механизма доступа к данным.

Здесь мы рассмотрим только уникальные свойства и методы класса `TCustomADODataSet`, обеспечивающие работу с ADO.

Соединение набора данных с хранилищем данных ADO осуществляется с помощью компонента `TADOConnection` (свойство `Connection`) или путем задания параметров соединения в уже рассмотренном свойстве `ConnectionString`.

Набор данных

Перед открытием набора данных необходимо установить тип блокировки, используемой при редактировании записей. Для этого применяется свойство

```
type TADOLockType = (ltUnspecified, ltReadOnly, ltPessimistic,
ltOptimistic, ltBatchOptimistic);
property LockType: TADOLockType;
```

- `ltUnspecified` — блокировка задается источником данных, а не компонентом;
- `ltReadOnly` — набор данных открывается 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 изненений в хранилище данных;
- `ltBatchOptimistic` — запись блокируется на время сохранения в хранилище данных при вызове метода `UpdateBatch`.

Примечание

Для того чтобы блокировка была установлена, свойство `LockType` должно быть модифицировано обязательно до открытия набора данных.

Набор данных открывается методом `Open` и закрывается методом `Close`. Также можно использовать свойство

```
property Active: Boolean;
```

Текущее состояние набора данных можно определить с помощью свойства

```
type
  TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stFetching);
  TObjectStates = set of TObjectState;
property RecordsetState: TObjectStates;
```

Набор данных в компонентах ADO основан на использовании объекта набора записей ADO, прямой доступ к этому объекту возможен при помощи свойства

```
property Recordset: _Recordset;
```

Поскольку 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 используемого курсора.

Местоположение курсора задается свойством

```
type TCursorLocation = (clUseServer, clUseClient);

property CursorLocation: TCursorLocation;
```

Курсор может находиться на сервере (`clUseServer`) или на стороне клиента (`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 разумно использовать только для небольших наборов данных, не загружающих канал связи с сервером.

При использовании 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 использовать значение `moMarshalAll`, разрешающее возврат серверу всех записей набора данных. В противном случае для ускорения работы компонента лучше применить значение `moMarshalModifiedOnly`, обеспечивающее возврат только модифицированных клиентом записей.

Тип курсора определяется свойством

```
TCursorType = (ctUnspecified, ctOpenForwardOnly, ctKeyset, ctDynamic,
ctStatic);

property CursorType: TCursorType;
```

- `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 стороне клиента (`CursorType = clUseClient`), то для него возможен только один тип — `ctStatic`.

Соответственно до и после каждого перемещения курсора в наборе данных вызываются методы-обработчики:

```
TRecordsetReasonEvent = procedure(DataSet: TCustomADODataSet; const Rea-
son: 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 равно числу записей, помещаемых в локальный буфер, и не может быть меньше 1. Очевидно, что при достаточно большом размере 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 комопненту память. ЕЕ можно определить 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. Общее число записей набора данных на данный момент возвращает свойство, доступное только для чтения

```
property RecordCount: Integer;
```

При достижении последней записи набора данных вызывается метод-обработчик

```
TEndOfRecordsetEvent = procedure (DataSet: TCustomADODataSet; var More-
Data: 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;
```

- `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 не сохраненные 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 из 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 параметр `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 выше свойству `RecordStatus`.

Перечислим возможные значения свойства :

- `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` — отображаются записи, полученные при последнем обновлении из источника данных;

- `fgPredicate` — отображаются только удаленные записи;
- `fgConflictingRecords` — отображаются модифицированные записи, при сохранении которых в хранилище данных возникла ошибка.

Для того чтобы групповая фильтрация выполнилась, необходимы два дополнительных условия. Во-первых, фильтрация должна быть включена — свойство `Filtered` должно иметь значение `True`. Во-вторых, для свойства `LockType` нужно установить значение `ltBatchOptimistic`.

```
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 вас 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 бы 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 бы находиться;
- `soBefore` — курсор устанавливается перед найденной записью.

Сортировка

Свойство

```
property Sort: WideString;
```

предоставляет простой способ сортировки по произвольному сочетанию полей. Оно должно содержать перечисленные через запятую имена нужных полей и признак прямого (ASC) или обратного (DESC) порядка сортировки:

```
ADODataSet.Sort := 'FirstField, SecondField DESC';
```

Если порядок сортировки не указан, по умолчанию задается прямой порядок.

Команда ADO

Для выполнения запросов к источнику данных любой компонент ADO содержит специальный объект-команду ADO.

При использовании компонентов-потомков класса TCustomADODataSet обычно нет необходимости применять объект-команду напрямую. Несмотря на то, что все реальное взаимодействие объекта-набора данных ADO с источником данных осуществляется через объект-команду, настройка и выполнение команды скрыты в свойствах и методах компонентов ADO. Тем не менее, в классе TCustomADODataSet доступ к объекту-команде 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, также расположенный на странице **ADO** Палитры компонентов.

Тип команды задается свойством

```
type
  TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc, cmdFile,
  cmdTableDirect);
property CommandType: TCommandType;
```

- cmdUnknown — тип команды неизвестен и будет определен источником данных;
- cmdText — текстовая команда, интерпретируемая источником данных (например, запрос SQL); текст должен быть составлен с учетом правил конкретного источника данных;
- cmdTable — команда получения набора данных таблицы из хранилища данных;
- cmdStoredProc — команда выполнения хранимой процедуры;

- `cmdFile` — команда получения набора данных, сохраненного в файле в формате конкретного источника данных;
- `cmdTableDirect` — команда получения набора данных таблицы напрямую, например, из файла таблицы.

Текст команды, представленный свойством

```
property CommandText: WideString;
```

обязательно должен быть согласован с ее типом.

Для органичения времени ожидания выполнения команды используется свойство

```
property CommandTimeout: Integer;
```

В компонентах наборов данных ADO команды выполняются для 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;
```

- `eoAsyncExecute` — асинхронное выполнение команды;
- `eoAsyncFetch` — асинхронное выполнение команды обновления набора данных;
- `eoAsyncFetchNonBlocking` — асинхронное выполнение команды обновления набора данных без установки блокировки;
- `eoExecuteNoRecords` — выполнение команды, не требующее возврата набора данных.

Групповые операции

Напомним, что наборы данных ADO используют на клиентской стороне локальный кэш для хранения данных и сделанных изменений. Благодаря наличию этого кэша и появилась возможность реализовать групповые операции. В этом 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 недоступность для других пользователей изменений, находящихся на стороне клиента. В этой ситуации могут возникать потери данных.

Для перевода набора данных в режим групповых операций необходимо соблюдение следующих условий.

- Набор данных должен использовать клиентский курсор:
  ```
  ADODataSet.CursorLocation := clUseClient;
  ```
- Курсор должен иметь тип `ctStatic`:
  ```
  ADODataSet.CursorType := ctStatic;
  ```
- У блокировки должно быть 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 применяется метод

```
procedure CancelBatch(AffectRecords: TAffectRecords = arAll);
```

Используемый в методах тип `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 не видимые из-за включенного фильтра), а также для всех вложенных наборов данных.

7.5.2. Параметры

Многие компоненты ADO, инкапсулирующие набор данных, должны обеспечивать возможность применения параметров запросов. Для этого в них используется специальный класс `TParameters`.

Для каждого параметра из коллекции класса `TParameters` создается отдельный класс `TParameter`.

Этот класс является наследником класса коллекции `TCollection` и содержит 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 порожденный 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 используется в процессе работы компонентов ADO, содержащих набор данных, поэтому для компонентов ADO в VCL был создан собственный класс параметров.

Класс *TParameters*

Главное назначение класса TParameters — хранение списка 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 метода

```
function AddParameter: TParameter;
```

который 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

```
function CreateParameter(const Name: WideString; DataType: TDataType;
Direction: TParameterDirection; Size: Integer; Value: OleVariant):
TParameter;
```

также создает новый параметр и определяет его свойства.

- `Name` — имя параметра.
- `DataType` — тип данных параметра, соответствующий типу поля таблицы БД (тип `TFieldType`).

- `Direction` — вид параметра, в дополнение к стандартным видам `dUnknown`, `pdInput`, `pdOutput`, `pdInputOutput`, класс `TParameterDirection` имеет дополнительный вид `pdReturnValue`, определяющий любое возвращаемое значение.
- `Size` — максимальный размер значения параметра.
- `Value` — значение параметра.

При работе с параметрами полезно вызывать их, используя имена, а не абсолютные индексы в списке. Для этого можно использовать метод

```
function ParamByName(const Value: WideString): TParameter;
```

Список параметров всегда должен соответствовать запросу или хранимой процедуре. Для обновления списка используется метод

```
procedure Refresh;
```

Также вы можете создать список параметров для запроса, не связанного с данным объектом параметров. Для этого используется метод

```
function ParseSQL(SQL: String; DoCreate: Boolean): String;
```

где `DoCreate` определяет, удалять ли перед анализом существующие параметры.

Класс *TParameter*

Класс `TParameter` содержит отдельный параметр.

Имя параметра определяется свойством

```
property Name: WideString;
```

Тип данных, которому должно соответствовать значение параметра, задается свойством

```
TDataType = TFieldType;
property DataType: TDataType;
```

Поскольку 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;
```

характеризует значение, присваемое параметру:

- `paSigned` — значение может быть символьным;
- `paNullable` — значение параметра может быть пустым;
- `paLong` — значение может содержать данные типа BLOB.

Вид параметра задается свойством

```
type TParameterDirection = (pdUnknown, pdInput, pdOutput, pdInputOutput,
pdReturnValue);
property Direction: TParameterDirection;
```

- `pdUnknown` — неизвестный вид, 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, используется в хранимых процедурах;
- `pdReturnValue` — параметр для передачи значения.

Если параметр должен передавать большие бинарные массивы (например, изображения или файлы), то значение параметра можно загрузить, используя методы

```
procedure LoadFromFile(const FileName: String; DataType: TDataType);
```

и

```
procedure LoadFromStream(Stream: TStream; DataType: TDataType);
```

7.5.3. Компонент *TADODataSet*

Компонент `TADODataSet` предназначен для представления набора данных из хранилища данных ADO. Он прост в использовании, имеет только несколько собственных свойств и методов и использует функции своего предка — класса `TCustomADODataSet`.

Это единственный содержащий набор данных компонент ADO, для которого свойства, позволяющие управлять командой ADO, доступны в Инспекторе объектов Delphi:

```
property CommandText: WideString;
```

и

```
property CommandType: TCommandType;
```

В результате компонент представляет собой гибкий инструмент, дающий возможность (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 лишь язык DML (Data Manipulation Language) (т. е. использовать только оператор `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 описанный в нем набор данных с визуальными компонентами отображения данных с помощью компонента `TDataSource`.

7.5.4. Компонент *TADOTable*

Компонент `TADOTable` обеспечивает использование в приложениях Delphi таблиц БД, подключенных с помощью провайдеров OLE DB. По своим функциональным возможностям и области применения он подобен стандартному табличному компоненту *(см. гл. 5)*.

Как вы уже знаете, в основе комопнента лежит использование команды 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).

Поскольку не все провайдеры ADO обеспечивают прямое использование таблиц БД, то для доступа к хранилищу 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".

7.5.5. Компонент *TADOQuery*

Компонент `TADOQuery` обеспечивает применение запросов SQL при работе с данными через ADO. По своей функциональности он подобен стандартному компоненту запроса *(см. гл. 5)*.

Текст запроса задается свойством

```
property SQL: TStrings;
```

Параметры запроса определяются свойством

```
property Parameters: TParameters;
```

Если запрос должен возвращать набор данных, для его открытия используется свойство

```
property Active: Boolean;
```

или метод

```
procedure Open;
```

В противном случае достаточно применить метод

```
function ExecSQL: Integer;
```

Число обработанных запросом записей возвращает свойство

```
property RowsAffected: Integer;
```

7.5.6. Компонент *TADOStoredProc*

Компонент `TADOStoredProc` позволяет использовать в приложениях Delphi, обращающихся к данным через ADO, хранимые процедуры. Он подобен стандартному компоненту хранимой процедуры *(см. гл. 5)*.

Имя хранимой процедуры задается свойством

```
property ProcedureName: WideString;
```

Для определения входных и выходных параметров используется свойство

```
property Parameters: TParameters;
```

Если процедура будет применяться без изменений многократно, имеет смысл заранее подготовить ее выполнение на сервере. Для этого свойству

```
property Prepared: Boolean;
```

присваивается значение `True`.

7.6. 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 (рис. 7.8). `TADOCommand` предназначен для выполнения команд, которые не возвращают наборы данных.

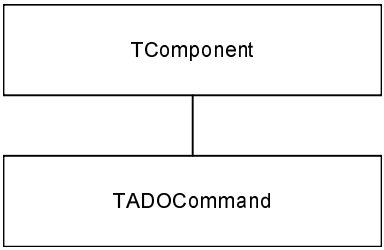


Рис. 7.8. Иерархия классов компонента `TADOCommand`

Поскольку компоненту `TADOCommand` нет необходимости обеспечивать работу набора данных, его непосредственным предком является класс `TComponent`. 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 ADO.

Текстовое представление выполняемой команды должно содержаться в свойстве

```
property CommandText: WideString;
```

однако команду можно задать и другим способом. Прямая ссылка на нужный объект-команду ADO может быть приведена в свойстве

```
property CommandObject: _Command;
```

Тип команды определяется свойством

```
type TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc,
cmdFile, cmdTableDirect);
property CommandType: TCommandType;
```

Тип `TCommandType` обладает избыточностью, так как используется и в классе `TCustomADODataSet`, где необходимо представлять все возможные виды команд. В этом свойстве не допустимы значения `cmdTable`, `cmdFile`, `cmdTableDirect`, а тип `cmdStoredProc` должен обозначать только те хранимые процедуры, которые не возвращают набор данных.

Если команда содержит текст запроса SQL, свойству `CommandType` следует присвоить 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 использовать 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 данных, то они не передаются в компонент.

При работе с компонентом `TADOConnection` желательно использовать значение `eoExecuteNoRecords`.

Для прерывания выполнения команды используется метод

```
procedure Cancel;
```

Текущее состояние команды можно определить с помощью свойства

```
type
  TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stFetching);
  TObjectStates = set of TObjectState;
property States: TObjectStates;
```

7.7. 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 методы интерфейса `Error`, предоставляемого многими методами других объектов ADO. Например, тип

```
TRecordsetEvent = procedure(DataSet: TCustomADODataSet; const Error:
Error; var EventStatus: TEventStatus) of object;
```

используемый для 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 она 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.

7.8. 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 для выбранных записей (рис. 7.9).

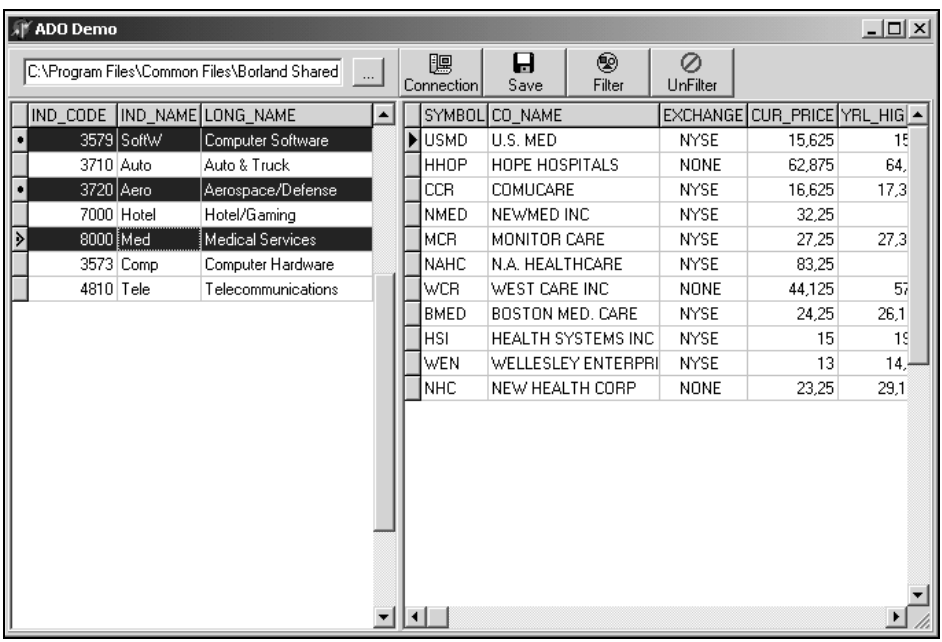


Рис. 7.9. Главное окно приложения ADO Demo

В качестве источника данных выберем файлы dBase, имеющиеся в демонстрационной базе данных Delphi ..\Program Files\Common Files\Borland Shared\Data. Используем в приложении две таблицы: INDUSTRY и MASTER. Они связаны между собой внешним ключом по полям IND_CODE и INDUSTRY соответственно.

Таблицу INDUSTRY можно редактировать, она содержится в компоненте `tblIndustry` типа `TADOTable` и отображается в левом компоненте `TDBGrid` (листинг 7.2). А таблица MASTER содержится в компоненте `tblMaster` и предназначена только для просмотра. Компоненты `tblIndustry` и `tblMaster` связаны отношением "один-ко-многим" при помощи свойств `MasterSource` и `MasterFields` соответственно.

Листинг 7.2. Секция `implementation` модуля `uMain` приложения ADO Demo

```
implementation

uses IniFiles, FileCtrl;

const sIniFileName: String = 'ADODemo.ini';
      sEmptyDefDB:  String = 'Не задан путь к базе данных';
      sEmptyFilter: String = 'Для задания фильтра не выбраны записи';

{$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',
  tblIndustry.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.
```

7.8.1. Соединение с источником данных

Для связывания приложения с источником данных используем компонент `TADOConnection` и настроим соединение, 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. 7.3). Как правило, он имеется в операционной системе, если вы не предпринимали специальных усилий по его удалению.

Далее, на странице **Connection** (см. рис. 7.4) выберем радиокнопку **Use data source name,** а 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 диалогового окна авторизации пользователя, ненужного для фалов dBase.

Свойство `DefaultDatabase` пока останется пустым. Мы применим его для указания пути к файлам базы данных, используя элементы 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 выполнять синхронно — соединение будет ожидать ответ на каждую команду.

Для свойства `Mode` установим значение `cmShareDenyNone`, которое 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 предназначена кнопка и однострочный редактор на панели управления. После выбора пути его значение заносится в переменную `DefDBStr` и в текст редактора `edDefDB`. Переменная используется для установки 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 ADODemo.ini и загружается при запуске приложения.

7.8.2. Групповые операции

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`.

7.8.3. Фильтрация

Для фильтрации записей в наборе данных `tblIndustry` используется метод `FilterOnBookmark`. Пользователь должен выбрать интересующие его записи в компоненте `dbgIndustry` (он работает в режиме `dgMultiSelect`). При нажатии кнопки `tbFilter` созданные в свойстве `SelectedRows` компонента `dbgIndustry` закладки передаются в массив `Bookmarks` типа `TVarRec`, который затем пересылается в качестве параметра методу `FilterOnBookmark` для фильтрации.

Массив `Bookmarks` служит здесь лишь промежуточным звеном для приведения типа закладок компонента `dbgIndustry` к типу параметра метода `FilterOnBookmark`.

7.8.4. Сортировка

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 ADO обеспечивает универсальный способ доступа к разнородным 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 программного обеспечения ADO.

Компонент `TADOConnection` позволяет соединяться с источниками данных с помощью провайдеров OLE DB. Компоненты `TADODataSet`, `TADOTable`, `TADOQuery`, `TADOStoredProc` обеспечивают использование наборов данных в приложении. Свойства и методы компонентов служат для создания полнофункциональных приложений.

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 III

Распределенные приложения баз данных

Глава 8. Технология DataSnap. Механизмы удаленного доступа

Глава 9. Сервер приложения

Глава 10. Клиент многозвенного распределенного приложения

Глава 8

Технология DataSnap. Механизмы удаленного доступа

В главах предыдущей части 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, так как 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.

Примечание

В версиях Delphi 4 и 5 эти компоненты объединялись под названием MIDAS (Multi-tier Distributed Applications Services — сервисы многозвенных распределенных приложений).

Палитра компонентов Delphi содержит специальную страницу **DataSnap**, на 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 темы:

- структура многозвенных приложений;
- механизм удаленного доступа к данным DataSnap;
- удаленные модули удаленных данных;
- компоненты-провайдеры;
- транспортные компоненты удаленных соединений DataSnap;
- вспомогательные компоненты — брокеры соединений.

8.1. Структура многозвенного приложения в Delphi

Появление многозвенной архитектуры приложений баз данных обусловлено 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 книги.

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 обслуживание такого программного обеспечения затруднено.

Многозвенная архитектура приложений БД призвана устранить перечисленные недостатки. Многозвенное приложение БД состоит из (рис. 8.1):

- "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, отображение 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.

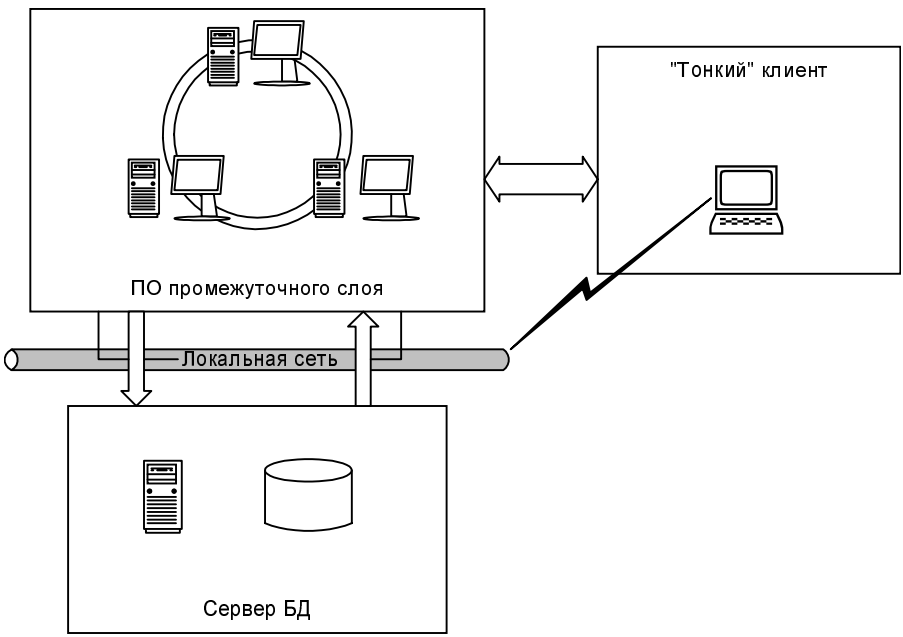


Рис. 8.1. Многозвенная архитектура приложений БД

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:

- "тонких" клиентов;
- 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 *гл. 9*, вопросы создания клиентского ПО — в *гл. 10*.

Сервер приложения взаимодействует с сервером БД, используя одну из технологий доступа к данным, реализованных в Delphi *(см. часть 2 "Технологии доступа к данным")*. Это технологии ADO, 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;
- MTS;
- WEB;
- SOAP;
- CORBA.

Удаленные клиентские приложения создаются с использованием специального набора компонентов, объединенных общим названием DataSnap. Эти компоненты описывают стандартные транспортные протоколы (DCOM, HTTP, CORBA, сокеты) и обеспечивают соединение удаленного клиентского приложения с сервером приложения. Кроме того, компоненты DataSnap предоставляют клиенту доступ к функциям сервера приложений благодаря использованию интерфейса `IAppServer` *(см. гл. 9)*.

Важную роль в разработке клиентских приложений играет компонент, содержащий клиентский набор данных. Его реализации также зависят от технологий доступа к данным и рассматриваются в *гл. 10*.

Наряду с перечисленными выше преимуществами, наличие дополнительного звена — сервера приложений — дает дополнительный выигрыш в решении задачи повышения надежности и эффективности системы.

Поскольку 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, так как вы можете организовать здесь авторизацию пользователей, да и реализовать любые другие функции безопасности, без прямого доступа к данным.

Кроме того, вы легко сможете использовать защищенные каналы передачи данных, например HTTPS.

8.2. Трехзвенное приложение в Delphi

Теперь рассмотрим составные части трехзвенного распределенного приложения в Delphi. Как говорилось ранее, в Delphi целесообразно разрабатывать клиентскую часть трехзвенного приложения и ПО промежуточного слоя — сервер приложения (рис. 8.2).

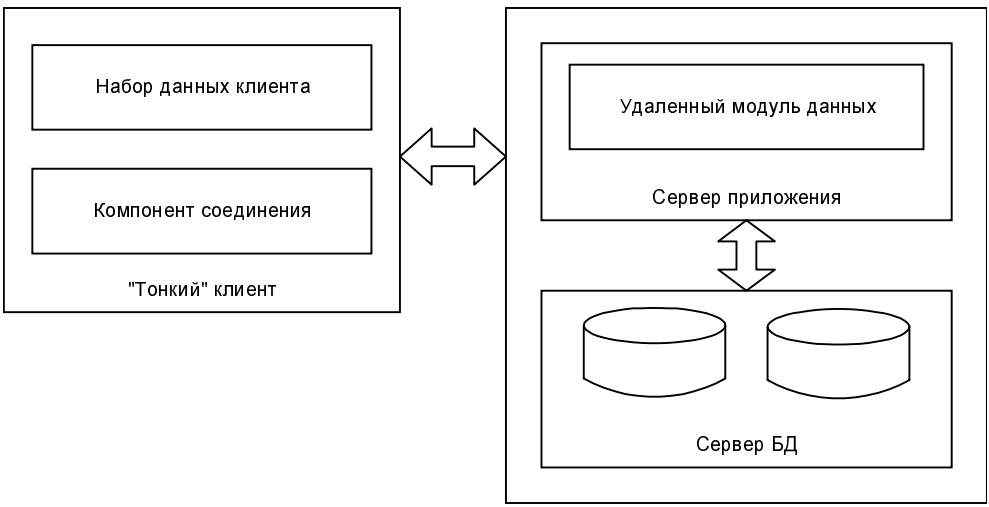


Рис. 8.2. Схема трехзвенного распределенного приложения

Части трехзвенных приложений проектируются 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 2 "Технологии доступа к данным")*.

Примечание

Разработку трехзвенных приложений целесообразно вести, используя в среде разработки группу проектов вместо одиночных проектов. Для этого используется утилита Project Manager (пункт меню **View | Project Manager**).

Передача данных между сервером приложений и клиентами осуществляется с помощью интерфейса `IAppServer`, предоставляемого сервером приложения. Этот интерфейс используют компоненты-провайдеры `TDataSetProvider` на стороне сервера и компоненты `TClientDataSet` на стороне клиента.

8.2.1. Сервер приложения

Сервер приложения содержит 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, включает в себя готовый удаленный сервер, который требует лишь настройки нескольких параметров и регистрации.

В состав Delphi входят пять типов удаленных модулей данных. Для их создания используйте страницы **Multitier**, **WebSnap** и **WebServices** Репозитария Delphi (рис. 8.3).

- **Remote Data Module** — удаленный модуль данных, инкапсулирующий сервер Автоматизации. Используется для организации соединений через DCOM, HTTP, сокеты. *Более подробную информацию см. в гл. 11 .*
- **Transactional Data Module** — удаленный модуль данных, инкапсулирующий сервер MTS (Microsoft Transaction Server). *Более подробную информацию о технологиях MTS иCOM+ см. в гл. 4.*
- **CORBA Data Module** — удаленный модуль данных, инкапсулирующий сервер CORBA.
- **Soap Server Data Module** — удаленный модуль данных, инкапсулирующий сервер SOAP (Simple Object Access Protocol).
- **WebSnap Data Module** — удаленный модуль данных, использующий Web-службы и Web-браузер в качестве сервера.

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. гл. 5)* удаленный модуль данных является платформой для размещения невизуальных компонентов доступа к данным и компонентов-провайдеров. Расположенные в нем компоненты соединений, транзакций и компоненты, содержащие наборы данных, обеспечивают трехзвенное приложение связью с сервером БД. Ими могут быть наборы компонентов для различных технологий доступа к данным.

Помимо удаленного модуля данных, неотъемлемой частью сервера приложения являются компоненты-провайдеры `TDataSetProvider`. В модуле данных с каждым компонентом, содержащим набор данных, предназначенный для передачи клиенту, должен быть связан компонент-провайдер.

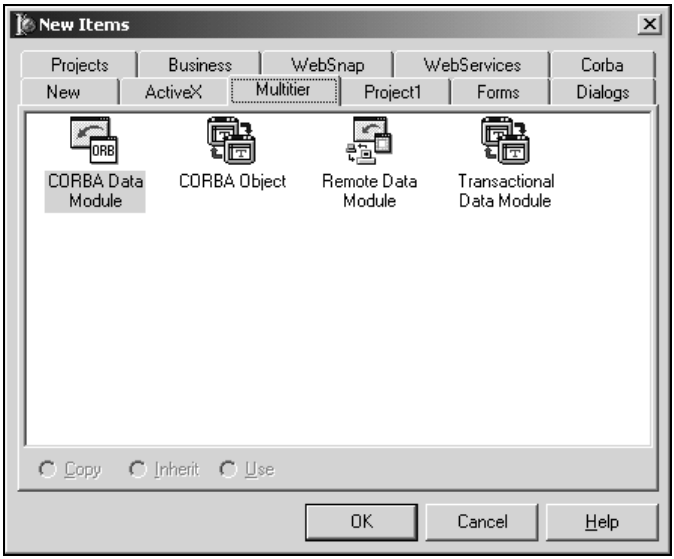


Рис. 8.3. Страницы Репозитария Delphi для выбора удаленных модулей данных

Для этого в удаленный модуль данных следует включить 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 использования методов этого интерфейса можно управлять пакетами пересылаемых данных на низком уровне.

Обычно такая задача перед разработчиками не стоит. Просто нужно знать, что компоненты, работающие с данными, как на стороне клиента, так и на стороне сервера, используют этот интерфейс. Впрочем, если вы собрались создавать собственный вариант технологии DataSnap, описание интерфейса `IProviderSupprot` вам пригодится *(см. гл. 9)*.

8.2.2. 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 удаленному клиентскому приложению необходимо обеспечить соединение с сервером приложения. Для этого используются компоненты соединений DataSnap:

- `TDCOMConnection` — использует DCOM;
- `TSocketConnection` — использует сокеты Windows;

- `TWebConnection` — использует HTTP;
- `TCORBAConnection` — использует соединение в рамках архитектуры CORBA.

Примечание

Компонент `TSOAPConnection` рассматривается отдельно, так как формально он используется в технологии WebSnap.

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 применяются компоненты `TClientDataSet`, работающие в режиме кэширования данных.

Для представления данных и создания пользовательского интерфейса в клиентском ПО предназначены стандартные компоненты со страницы **Data Controls** Палитры компонентов.

Подробнее о разработке клиентского ПО для распределенных многозвенных приложений БД рассказывается в гл. 10.

8.3. Механизм удаленного доступа к данным DataSnap

Для передачи пакетов данных от клиентского набора данных компоненту-провайдеру и обратно (см. рис. 8.2) между клиентом и сервером должен существовать некий транспортный канал, обеспечивающий физическую передачу данных. Для этого могут использоваться разнообразные транспортные протоколы, поддерживаемые операционной системой.

Различные типы соединений, позволяющие настроить транспортный канал и начать передачу и прием данных, содержатся в нескольких компонентах DataSnap. Для создания соединения с использованием того или иного транспортного протокола на стороне клиента разработчику достаточно перенести соответствующий компонент на форму и правильно настроить несколько свойств.

Такой 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, HTTP, CORBA.

8.3.1. Компонент *TDCOMConnection*

Компонент `TDCOMConnection` предоставляет транспортный канал 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 обязательно должны быть заданы. Только имя сервера или только его GUID не обеспечит правильный доступ к удаленному объекту COM.

Открытие и закрытие соединения осуществляется 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 IAppServer: 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 приложения.

Методы-обработчики компонента `TDCOMConnection` представлены в табл. 8.1.

Таблица 8.1. *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
`type TGetUsernameEvent = procedure(Sender: TObject; var Username: string) of object;` `property OnGetUsername: TGetUsernameEvent;`	Вызывается непосредственно перед появлением диалогового окна удаленной авторизации пользователя. Для этого свойство `LoginPrompt` должно иметь значение `True`. Параметр `Username` может содержать имя пользователя по умолчанию, которое появится в соответствующем поле
`type TLoginEvent = procedure(Sender:TObject; Username, Password: string) of object;` `property OnLogin: TLoginEvent;`	Вызывается после открытия соединения, если свойство `LoginPrompt` имеет значение True. Парметры `Username` и `Password` содержат имя пользователя и пароль, введенные при авторизации

8.3.2. Компонент *TSocketConnection*

Компонент `TSocketConnection` обеспечивает соединение клиента с сервером приложения за счет использования сокетов TCP/IP (рис. 8.4). Для успешного открытия соединения на стороне сервера должен работать сокет-сервер (приложение ScktSrvr.exe).

Для успешного соединения свойство

```
property Host: String;
```

должно содержать имя компьютера-сервера. Дополнительно в свойстве

```
property Address: String;
```

необходимо указать IP адрес сервера.

Для открытия соединения должны быть заданы оба этих свойства.

Свойство

```
property Port: Integer;
```

устанавливает номер используемого порта. По умолчанию это порт 211, но разработчик волен изменить порт, например, для использования различными категориями пользователей или для создания защищенного канала.

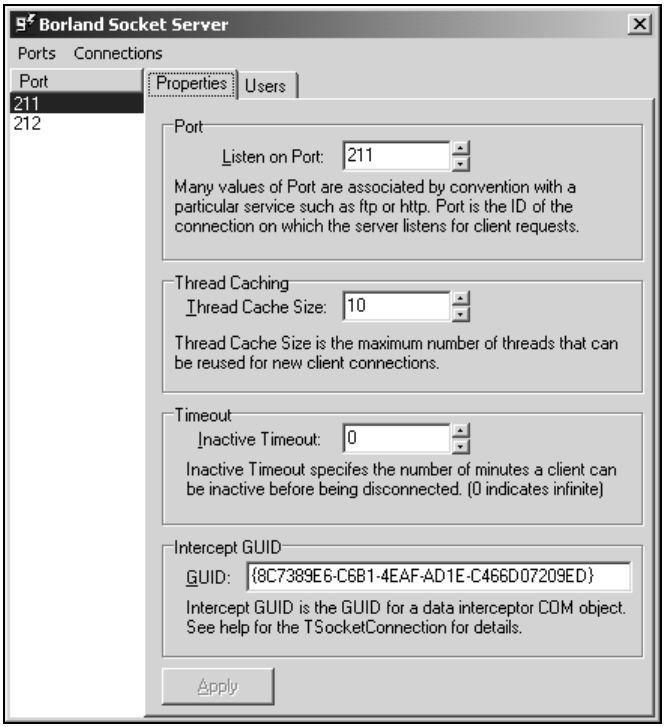


Рис. 8.4. Сокет-сервер ScktSrvr.exe

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, устанавливается автоматически, хотя его можно задать и вручную.

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 используется свойство

```
property InterceptName: string;
```

содержащее программный идентификатор объекта COM, обеспечивающего шифрование/дешифрование данных в канале (рис. 8.5), 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 может быть шифрование, сжатие, обработка шумов и т. д.

Примечание

Создание объекта COM, отвечающего за дополнительную обработку данных в канале, ложится на плечи разработчика. Объект-перехватчик должен поддерживать стандартный интерфейс `IDataIntercept`.

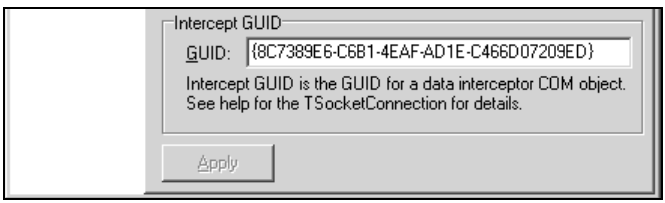


Рис. 8.5. Регистрация объекта-перехватчика COM на сокет-сервере

Естественно, на стороне сервера нужно зарегистрировать объект COM, выполняющий обратную операцию. Для этого также используется сокет-сервер (см. рис. 8.5). 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 использовать 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.8.1).

8.3.3. Компонент *TWebConnection*

Компонент `TWebConnection` предоставляет клиенту соединение на основе транспортного протокола HTTP. Для работы компонента на клиентском компьютере должна быть зарегистрирована библиотека wininet.dll. Обычно это не требует специальных усилий, так как этот файл уже есть 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;
```

хранит имя используемого прокси-сервера.

В заголовок сообщений HTTP можно поместить имя приложения. Для этого используется 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 во время выполнения.

Методы-обработчики событий компонента `TWebConnection` полностью совпадают с методами-обработчиками компонента `TDCOMConnection` (см. табл.8.1).

8.3.4. Компонент *TCORBAConnection*

Компонент `TCORBAConnection` обеспечивает доступ клиентского приложения к серверу CORBA. Настройка соединения с сервером осуществляется одним единственным свойством

```
type TRepositoryId = type string;
property RepositoryId: TRepositoryId;
```

в котором через слеш указывается имя сервера и имя удаленного модуля данных. Например, для сервера `CORBAServer` и модуля `CORBAModule` свойство `RepositoryId` должно иметь следующее значение:

```
CORBAServer/ CORBAModule
```

Или же, адрес сервера можно представить в нотации IDL (Interface Definition Language):

```
IDL: CORBAServer/ CORBAModule:1.0
```

Свойство

```
property HostName: string;
```

должно содержать имя компьютера сервера или его IP адрес. Если свойство не имеет значения, компонент `TCORBAConnection` соединяется с первым найденным сервером, имеющим заданные свойством `RepositoryId` параметры.

Имя сервера CORBA хранится в свойстве

```
property ObjectName: string;
```

Соединение открывается и закрывается при помощи свойства

```
property Connected: Boolean;
```

Аналогичные операции выполняют методы

```
procedure Open;
procedure Close;
```

Если соединение по какой-либо причине не открывается, для предотвращения зависания приложения можно использовать свойство

```
property Cancelable: Boolean;
```

Если оно имеет значение `True`, после ожидания установки соединения более одной секунды соединение прерывается. При этом может быть определен метод-обработчик

```
type TCancelEvent = procedure(Sender: TObject; var Cancel: Boolean;
var DialogMessage: string) ofobject;
property OnCancel: TCancelEvent;
```

который вызывается перед разрывом соединения.

После открытия соединения с помощью свойства

```
property AppServer: Variant;
```

или метода

```
function GetServer: IAppServer; override;
```

клиенту предоставляется интерфейс `IAppServer`.

Методы-обработчики компонента `TCORBAConnection` унаследованы (за исключением представленного выше метода `OnCancel`) от класса-предка `TCustomConnection`. Их описание см. в табл. 8.1.

8.4. Вспомогательные компоненты — брокеры соединений

В состав компонентов DataSnap входит ряд дополнительных компонентов, облегчающих работу с соединениями между удаленными клиентами и сервером приложений. Рассмотрим их.

8.4.1. Компонент *TSimpleObjectBroker*

Компонент `TSimpleObjectBroker` включает в себя список серверов, доступных клиентам 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. В этом случае при повторном открытии соединения имя сервера будет запрашиваться из списка компонента `TSimpleObjectBroker`.

Список серверов задается свойством

```
property Servers: TServerCollection;
```

На этапе разработки этот список заполняется специализированным редактором (рис. 8.6), который вызывается при щелчке на кнопке свойства в Инспекторе объектов.

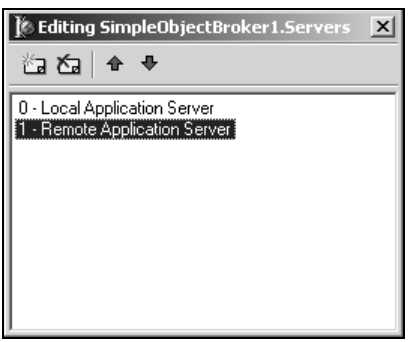


Рис. 8.6. Редактор списка серверов компонента `TSimpleObjectBroker`

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 задать имя сервера для представления в списке серверов:

```
property DisplayName: String;
```

Есть возможность сделать доступной или недоступной для выбора запись о сервере. Для этого используется свойство

```
property Enabled: Boolean;
```

После неудачной попытки при подключении использовать данную запись списка свойству

```
property HasFailed: Boolean;
```

присваивается значение `True`, и в дальнейшем эта запись не применяется.

Свойство

```
property Port: Integer;
```

содержит номер порта, используемого при подключении к серверу.

При выполнении соединения значения этих свойств подставляются в соответсвующие 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.

Метод

```
function GetComputerForProgID(const ProgID): string; override;
```

возвращает имя компьютера, на котором зарегистрирован сервер с именем, содержащемся в параметре `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.

8.4.2. 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. Для этого используется его свойство

```
property AppServer: IAppServer;
```

или метод

```
function GetServer: IAppServer; override;
```

8.4.3. 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. гл. 9)*. Компонент `TSharedConnection` используется 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 имя дочернего удаленного модуля данных определяется свойством

```
property ChildName: string;
```

Если интерфейс главного удаленного модуля данных настроен правильно, то в списке выбора этого 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. 8.1).

8.4.4. 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 транспортного протокола HTTP на сокеты TCP/IP) нет необходимости переопределять значение свойства `RemoteServer` всех компонентов `TClientDataSet`, а достаточно задать 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 приведенным в табл. 8.1.

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 используются компоненты DataSnap и удаленные модули данных. Все эти инструменты реализованы для различных типов транспортных протоколов.

Кроме того, в трехзвенных приложениях применяются компоненты-провайдеры `TDataSetProvider` и компоненты `TClientDataSet`, инкапсулирующие наборы данных на клиентской стороне.

Глава 9

Сервер приложения

Многозвенные распределенные приложения обеспечивают эффективный доступ удаленных клиентов к базе данных, так как 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, добиваясь сбалансированной нагрузки сервера БД;
- может содержать часть бизнес-логики распределенного приложения, тем самым облегчая алгоритм клиентских приложений и превращая их в "тонких" клиентов.

Delphi позволяет разрабатывать серверы приложений на основе использования ряда технологий:

- Web;
- Автоматизация;
- MTS;
- CORBA;
- SOAP.

В этой главе рассматриваются следующие темы:

- 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.

9.1. Структура сервера приложения

Итак, сервер приложения — это ПО промежуточного слоя трехзвенного распределенного приложения (рис. 9.1). Его основой является удаленный модуль данных. В Delphi предусмотрено использование удаленных модулей данных пяти типов.

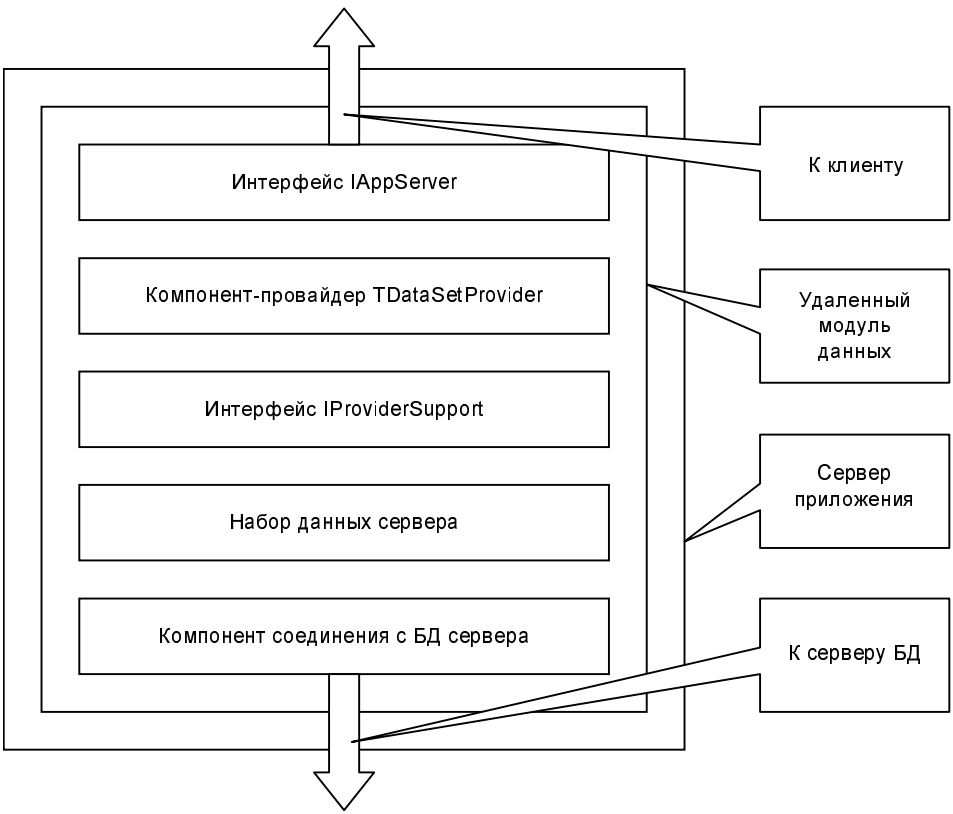


Рис. 9.1. Структура сервера приложения

Далее в этой главе мы подробно рассмотрим вопросы использования удаленных модулей данных, инкапсулирующих функции серверов Автоматизации. Другие типы удаленных модулей данных обсуждаются в следующих частях этой книги.

Каждый удаленный модуль данных содержит интерфейс `IAppServer`, методы которого используются в механизме удаленного доступа клиентов к серверу БД *(см. гл. 8)*.

Для обмена данными с сервером БД модуль данных может содержать некоторое количество компонентов доступа к данным (компонентов соединений и компонентов, инкапсулирующих набор данных).

Удаленный модуль данных для обеспечения передачи данных клиентам обязательно должен содержать необходимое количество компонентов `TDataSetProvider`, каждый из которых нужно связать 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 действий.

1. Создать новый проект, выбрав в качестве типа проекта обычное приложение (пункт меню **File | New | Application**) и сохранить его.
2. Выбрать из Репозитария Delphi тот или другой в зависимости от используемой технологии тип удаленного модуля данных. Удаленные модули данных располагаются на страницах **Multitier**, **WebSnap** и **WebServices**.
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 II "Технологии доступа к данным")* в зависимости от используемого 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 разработать для потомка интерфейса `IAppServer`, применяемого 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 разработки сервера приложения.

9.2. Интерфейс *IAppServer*

Интерфейс `IAppServer` является основой 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. 9.1).

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 есть как в компонентах удаленного соединения, так и в компонентах, содержащих клиентский набор данных.

По умолчанию интерфейс является не сохраняющим состояние (stateless). Это означает, что вызовы методов интерфейса независимы и не привязаны к предыдущему вызову. Таким образом, интерфейс `IAppServer` не имеет свойств, которые бы хранили информацию о состоянии между вызовами.

Обычно разработчику незачем использовать методы интерфейса напрямую, однако его значение для многозвенных приложений трудно переоценить. И при необходимости вносить изменения в механизм удаленного доступа интерфейс понадобится, так или иначе.

Методы интерфейса `IAppServer` представлены в табл. 9.1.

Таблица 9.1. *Методы интерфейса* `IAppServer`

Объявление	Описание
`function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer; var OwnerData: OleVariant): OleVariant; safecall;`	Передает полученные от клиентского набора данных изменения компоненту-провайдеру, заданному 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`

Таблица 9.1 *(продолжение)*

Объявление	Описание
	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
`function AS_DataRequest (const ProviderName: WideString; Data: OleVariant): OleVariant; safecall;`	Генерирует событие `OnDataRequest` для провайдера, указанного в `ProviderName`
`procedure AS_Execute (const ProviderName: WideString; const CommandText: WideString; var Params: OleVariant; var OwnerData: OleVariant); safecall;`	Выполняет запрос или хранимую процедуру, заданные параметром `CommandText`, для провайдера, указанного в параметре `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. Эта информация принимается во внимание только в том случае, если для провайдера в параметре `Options` включен режим `poAllowCommandText`. Парметры запроса или процедуры размещаются 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

Таблица 9.1 *(окончание)*

Объявление	Описание
`function AS_RowRequest(const ProviderName: WideString; Row: OleVariant; RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;`	Возвращает определяемую параметром `Row` запись набора данных, предоставляемого провайдером `ProviderName`. Параметр `RequestType` содержит значение типа `TfetchOptions`

Большинство методов интерфейса использует параметры `ProviderName` и `OwnerData`. Первый определяет имя компонента-провайдера, а второй содержит набор параметров, передаваемых для применения в методах-обработчиках.

Внимательный читатель заметил, что использование метода `AS_GetRecords` подразумевает сохранение информации при работе интерфейса, так как метод возвращает записи, начиная с текущей, хотя интерфейс `IAppServer` имеет тип stateless. Поэтому перед использованием этого метода рекомендуется обновлять набор данных клиента.

Тип

```
TFetchOption = (foRecord, foBlobs, foDetails);
TFetchOptions = set of TFetchOption;
```

используется в параметре `RequestType` метода `AS_RowRequest`:

- `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.

9.3. Удаленные модули данных

Удаленный модуль данных является основой сервера приложения (см. рис. 9.1) для трехзвенного распределенного приложения.

Во-первых, удаленный модуль данных инкапсулирует тот или иной, в зависимости от реализации, удаленный сервер.

Во-вторых, удаленный модуль данных содержит интерфейс `IAppServer`, обеспечивая тем самым выполнение функций сервера и обмен данными с удаленными клиентами.

В-третьих, он выполняет функции обычного модуля данных — в нем можно размещать компоненты доступа к данным.

В зависимости от используемой технологии, в Delphi можно применять удаленные модули данных пяти типов.

- **Remote Data Module**. Класс `TRemoteDataModule` инкапсулирует сервер Автоматизации.
- **Transactional Data Module**. Класс `TMTSDataModule` является потомком класса `TRemoteDataModule` и к функциям обычного сервера Автоматизации добавляет возможности MTS.
- **WebSnap Data Module**. Класс `TWebDataModule` создает сервер приложения, использующий возможности Internet-технологий.
- **Soap Server Data Module**. Класс `TSOAPDataModule` инкапсулирует сервер SOAP.
- **CORBA Data Module**. Класс `TCORBADataModule` является потомком класса `TRemoteDataModule` и реализует функции сервера CORBA.

Ниже мы рассмотрим процесс разработки сервера приложения на основе удаленного модуля данных `TRemoteDataModule`. Остальные модули данных (за исключением удаленного модуля данных для CORBA) подробно рассматриваются далее в этой книге.

9.3.1. Удаленный модуль данных для сервера Автоматизации

Для создания удаленного модуля данных `TRemoteDataModule` используется Репозитарий Delphi (команда пункта меню **File | New | Other**). Значок класса `TRemoteDataModule` находится на странице **Multitier**. Перед созданием экземпляра удаленного модуля данных появляется диалоговое окно (рис. 9.2), в котором необходимо задать три параметра.

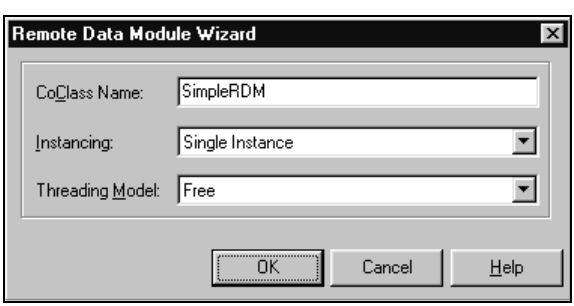


Рис. 9.2. Мастер создания удаленного модуля данных `TRemoteDataModule`

Строка **CoClass Name** должна содержать имя нового модуля данных, которое будет также использовано для именования нового класса, создаваемого для поддержки этого 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 формироваться 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 разработке 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 сформирует для нового модуля данных исходный код, представленный в листинге 9.1.

Листинг 9.1. 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 на то, 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 использование интерфейсов.

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 использовать этот метод, так как 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. Интерфейс `ISimpleRDM`, созданный для удаленного модуля данных `SimpleRDM`, представлен в листинге 9.2. Для удобства из листинга удалены автоматически добавляемые комментарии.

Листинг 9.2. 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, интерфейс `ISimpleRDM` является потомком интерфейса `IAppServer`, рассмотренного ранее.

Поскольку удаленный модуль данных реализует сервер Автоматизации, дополнительно к основному дуальному интерфейсу `ISimpleRDM` автоматически создан интерфейс диспетчеризации `ISimpleRDMDisp`. При этом для интерфейса диспетчеризации сформированы 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;
```

используется при работе с локальным (local) и внутренним (in-process) серверами.

Метод

```
class function CreateRemote(const MachineName: string): ISimpleRDM;
```

применяется в составе удаленного сервера.

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, так как 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.

9.3.2. 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. гл. 10)* будут доступны имена всех удаленных модулей данных, включенных в состав сервера приложения.

Однако в этом случае для каждого модуля понадобится собственный компонент соединения. Если это нежелательно, можно использовать компонент `TsharedConnection` *(см. гл. 8)*, но в интерфейсы удаленных модулей данных придется 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 особенности создания удаленных модулей данных в этом случае. 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 стороне клиента. 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. гл. 8)*.

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.

9.4. 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. 9.3).

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;
```

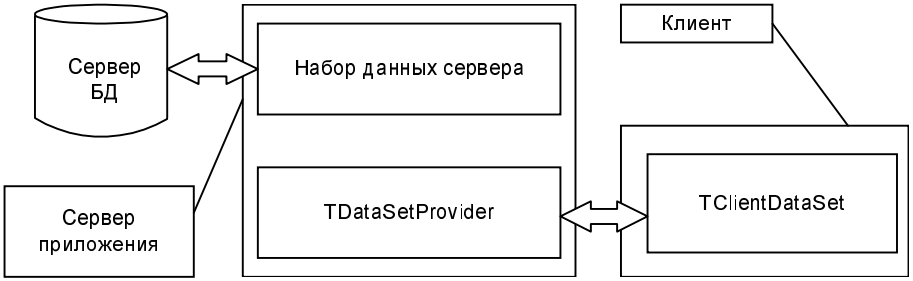


Рис. 9.3. Взаимодействие компонента-провайдера с клиентом

Далее, если соединение в клиентском приложении настроено правильно, то в Инспекторе объектов, в списке выбора свойства `ProviderName` компонента `TclientDataSet`, появляются имена всех компонентов-провайдеров сервера приложения. Если связать клиентский набор данных с компонентом-провайдером, а затем открыть этот набор, в него 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;
```

позволяет использовать 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;
```

- `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 следующих свойств для объектов полей: `Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, `DisplayValues`.
- `poCascadeDeletes` — включает автоматическое удаление на стороне сервера подчиненных записей в отношении "один-ко-многим", если главная запись была удалена в клиентском наборе данных.
- `poCascadeUpdates` — включает автоматическое обновление на стороне сервера подчиненных записей в отношении "один-ко-многим", если главная запись была изменена в клиентском наборе данных.
- `poReadOnly` — включает режим "только для чтения" для набора данных сервера.
- `poAllowMultiRecordUpdates` — включает режим внесения изменений в несколько записей одновременно. Иначе все записи изменяются последовательно, одна за другой.
- `poDisableInserts` — запрещает клиенту вставлять 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 клиентом порядка сортировки записей. Если этот параметр отключить, возможны ошибки отображения набора данных, проявляющиеся в появлении двойных записей.

Методы-обработчики компонента-провайдера данных представлены в табл. 9.2.

Таблица 9.2. *Методы-обработчики событий компонента* `TDataSetProvider`

Объявление	Описание
`Property AfterApplyUpdates: TremoteEvent;`	Вызывается после сохранения в наборе данных сервера изменений, переданных от клиента
`Property AfterExecute: TRemoteEvent;`	Вызывается после выполнения на сервере запроса SQL или хранимой процедуры
`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
`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 переданных от клиента изменений в наборе данных сервера
`property BeforeExecute: TRemoteEvent;`	Вызывается перед выполнением на сервере запроса SQL или хранимой процедуры
`property BeforeGetParams: TRemoteEvent;`	Вызывается перед тем, как компонент-провайдер сформировал набор параметров набора данных сервера для их передачи клиенту

Таблица 9.2 *(окончание)*

Объявление	Описание
`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

9.5. 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 используемой 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, наследников класса `TdataSet`, инкапсулирующих набор данных.

9.6. Регистрация сервера приложения

Сервер приложения должен быть зарегистрирован на компьютере сервера, для того, чтобы клиент мог его "увидеть". Процесс регистрации зависит от используемой технологии. Регистрация серверов MTS, Web и SOAP рассматривается далее в этой книге.

Здесь же мы остановимся на регистрации сервера приложения, использующего удаленный модуль 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 диалоговом окне команды пункта меню **Run | Parameters** (рис. 9.4).

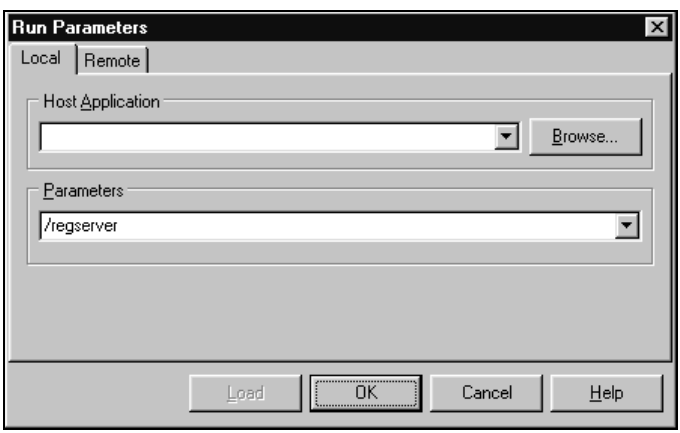


Рис. 9.4. Диалоговое окно для установки параметров запуска приложения

Для удаления регистрации используется ключ /unregserver, но только в командной строке.

Для регистрации динамических библиотек используется ключ /regsvr32.

9.7. 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 (табл. 9.3).

Таблица 9.3. *Файлы проекта* `SimpleAppSrvr`

Файл	Назначение
uSimpleAppSrvr.pas	Стандартный файл проекта
SimpleAppSrvr_TLB.pas	Библиотека типов, которая 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 *гл. 10*.

9.7.1. Главная форма сервера приложения

При первом запуске сервера приложения необходимо настроить соединение ADO, которое используется для доступа к источнику данных в модулях данных. На главной форме `fmMain` сервера приложения (файл uSimpleAppSrvr.pas) необходимо выбрать реальный путь к базе данных, нажав кнопку стандартного диалога выбора файла (рис. 9.5). Выбранный файл отображается в однострочном редакторе `edDataPath`.

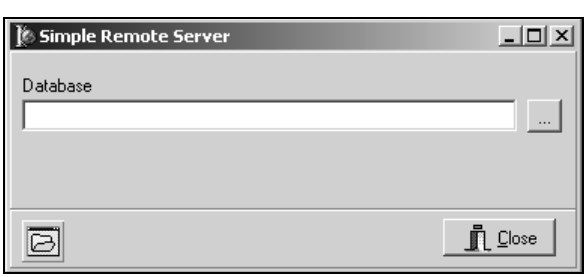


Рис. 9.5. Главная форма сервера приложения SimpleAppSrvr

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.

9.7.2. Главный удаленный модуль данных

Добавим в проект новый удаленный модуль данных, используя для этого Репозитарий Delphi (см. рис. 8.3). Затем в появившемся диалоговом окне (см. рис. 9.2) 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 9.1).

Одновременно с удаленным модулем данных автоматически создается библиотека типов и в ней дуальный интерфейс `ISimpleRDM` и интерфейс диспетчеризации `ISimpleRDMDisp` (см. листинг 9.2).

Примечание

Для каждого вновь созданного интерфейса автоматически назначается GUID.

Разместим в модуле `SimpleRDM` компоненты для доступа к файлам демонстрационной базы данных Microsoft Access, хранящейся в папке ..\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb, — это компонент `TADOConnection`, обеспечивающий соединение, и три табличных компонента `TADOTable`, инкапсулирующие наборы данных из таблиц Orders, Customer, Employee. Для доступа к данным используем провайдер OLE DB.

Перед открытием соединения вызывается метод-обработчик:

```
procedure TSimpleRDM.conMastAppBeforeConnect(Sender: TObject);
begin
 if fmMain.edDataPath.Text <> ''
  then conMastApp.ConnectionString := 'Provider =
  Microsoft.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 полученных от клиента изменений в наборе данных связанного компонента. Вместо этого данные напрямую сохраняются в базе данных. Это увеличивает быстродействие приложения.

9.7.3. 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. 9.6).

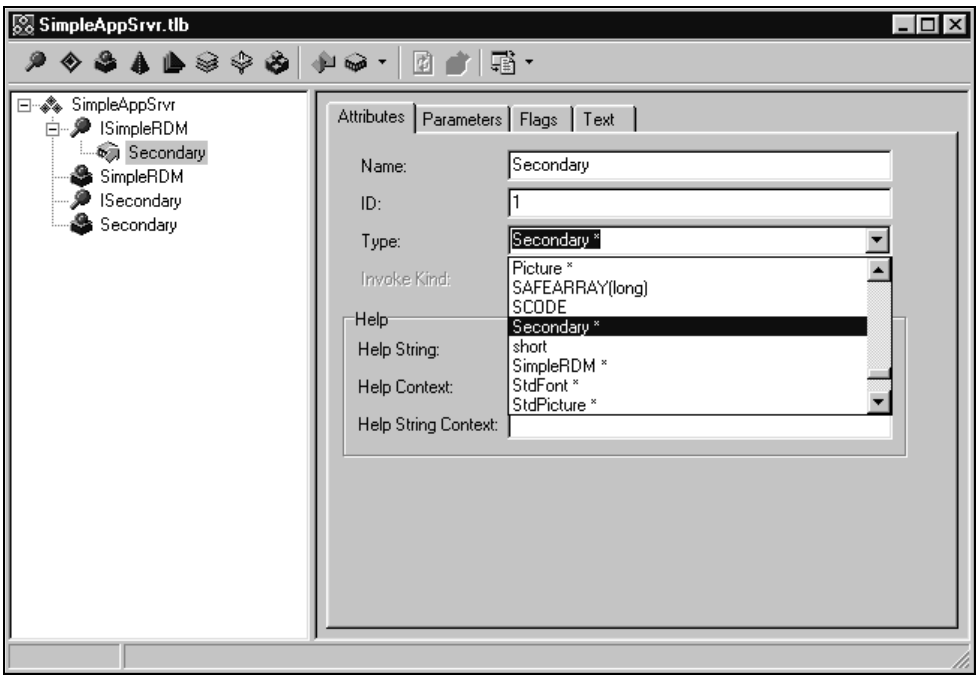


Рис. 9.6. Библиотека типов сервера приложения `SimpleAppSrvr`

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 Редактора библиотеки типов (см. рис. 9.6).

После обновления исходного кода библиотеки типов (кнопка **Refresh Implementation**) описание нового свойства и метода интерфейса `ISimpleRDM`

появится в файле SimpleAppSrvr_TLB.PAS. 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, используемая в этом примере, 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 использование вложенных наборов данных *(см. гл. 10)*.

9.7.4. Регистрация сервера приложения

Сервер приложения готов, остался последний этап — его регистрация. Для этого достаточно запустить исполняемый файл проекта на компьютере, который будет использоваться для работы 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 большую часть бизнес-логики распределенного приложения.

В Delphi сервер приложения создается на основе удаленных модулей данных, реализация которых различается в разных технологиях удаленного доступа. Удаленные модули данных содержат интерфейс `IAppServer`. Непосредственный доступ к данным обеспечивают компоненты-провайдеры `TDataSetProvider` при помощи интерфейса `IProviderSupport`.

Глава 10

Клиент многозвенного распределенного приложения

Клиентское ПО в распределенном многозвенном приложении имеет особенности архитектуры, продиктованные его ролью — ведь большая часть бизнес-логики и функций обработки данных сосредоточены в сервере приложений *(см. гл. 9)*. Такая схема призвана обеспечить более высокую эффективность обработки запросов многочисленных удаленных клиентов, а также упростить обслуживание клиентского ПО. Клиенты, выполняющие лишь необходимый минимум операций, называются "тонкими".

Клиенты многозвенных приложений обеспечивают выполнение следующих функций:

- соединение с сервером приложений, прием и передачу данных;
- отображение данных средствами пользовательского интерфейса;
- простейшие операции редактирования;
- сохранение локальных копий данных.

При разработке клиентских частей многозвенных приложений в Delphi используются компоненты DataSnap *(см. гл. 8)*, 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. гл. 5)*.

В этой главе рассматриваются следующие темы:

- 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.

10.1. Структура клиентского приложения

По структуре клиентское приложение подобно обычному приложению баз данных, описанному в *гл. 5*.

Соединение клиента с сервером приложений осуществляется специализированными компонентами DataSnap. В зависимости от потребностей в той или иной транспортной технологии разработчик может использовать следующие компоненты:

- `TDCOMConnection` — DCOM;
- `TSocketConnection` — сокеты TCP/IP;
- `TWEBConnection` — Web;
- `TCORBAConnection` — CORBA.

Эти компоненты взаимодействуют с удаленным модулем данных, входящим в состав сервера приложений, при помощи методов интерфейса `IAppServer`.

Кроме того, в клиентском приложении могут использоваться дополнительные, определенные разработчиком, методы интерфейса удаленного модуля данных, унаследованного от интерфейса `IAppServer`. *Более подробную информацию об этих компонентах и способах их настройки на удаленный сервер приложения см. в гл. 9.*

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 и связанные с визуальными компонентами отображения данных посредством компонентов `TDataSource`.

Очевидно, что набор данных сервера должен быть скопирован клиентским приложением в некий локальный буфер. При этом должен использоваться 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 методов, класс `TClientDataSet` содержит ряд дополнительных функций, облегчающих управление данными.

Примечание

Подобно обычному приложению БД, в "тонком" клиенте для размещения невизуальных компонентов доступа к данным необходимо использовать модули данных.

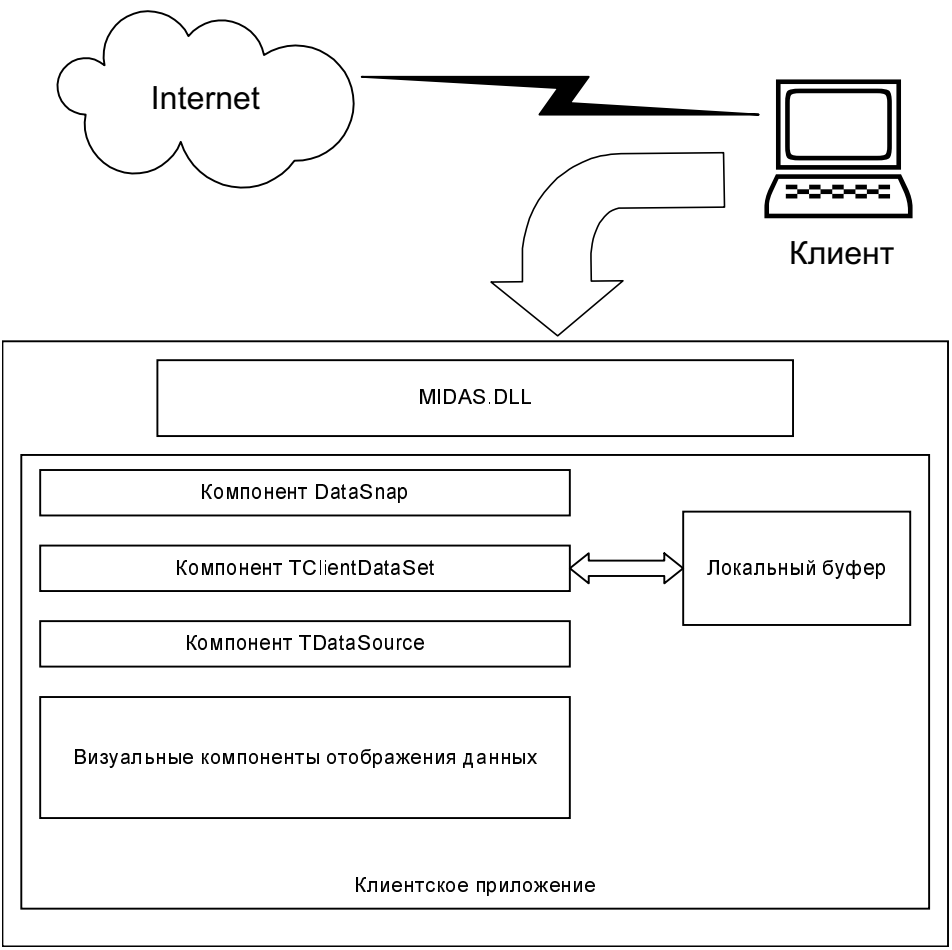


Рис. 10.1. Структура клиентской части многозвенного приложения Delphi

Для получения набора данных сервера компонент `TClientDataSet` взаимодействует с компонентом `TDataSetProvider`, используя методы интерфейса `IProviderSupport` *(см. гл. 9)*.

По существу, все уникальные функции клиентского приложения сосредоточены в компоненте `TClientDataSet`, изучением которого мы и займемся далее в этой главе. В остальном клиентское приложение ничем не отличается от обычного приложения БД, и для его разработки могут применяться стандартные методы.

10.2. Клиентские наборы данных

В Палитре компонентов Delphi представлено несколько компонентов, инкапсулирующих клиентский набор данных. В то же время при разработке настоящих удаленных клиентских приложений почему-то применяется компонент `TClientDataSet`. Внесем ясность в этот вопрос. Итак, помимо компонента `TClientDataSet`, расположенного на странице **Data Access**, представлены еще три компонента.

- Компонент `TSQLClientDataSet` разработан для технологии доступа к данным dbExpress и, по существу, является единственным полноценным средством для работы с набором данных в рамках этой технологии.
- Компонент `TBDEClientDataSet` используется при создании приложений для процессора BDE.
- Компонент `TIBClientDataSet` применяется в технологии доступа к данным сервера InterBase — InterBase Express.

Все перечисленные компоненты произошли от общего предка — класса `TCustomClientDataSet` (рис. 10.2) и обеспечивают локальное кэширование данных и взаимодействие с серверным набором данных при посредстве интерфейса `IProviderSupport`.

Основное различие между компонентом `TClientDataSet` и другими клиентскими компонентами заключается в том, что первый предназначен для использования с *внешним* компонентом-*провайдером данных*. Следовательно, он может взаимодействовать с удаленным провайдером данных.

Остальные перечисленные компоненты инкапсулируют *внутренний провайдер данных*, предоставляя тем самым для использования в рамках соответствующих технологий доступа к данным эффективный механизм локального кэширования данных. Применение внутреннего провайдера данных обеспечивает общий класс-предок `TCustomCachedDataSet`.

Для этого он имеет защищенное свойство

```
property Provider: TDataSetProvider;
```

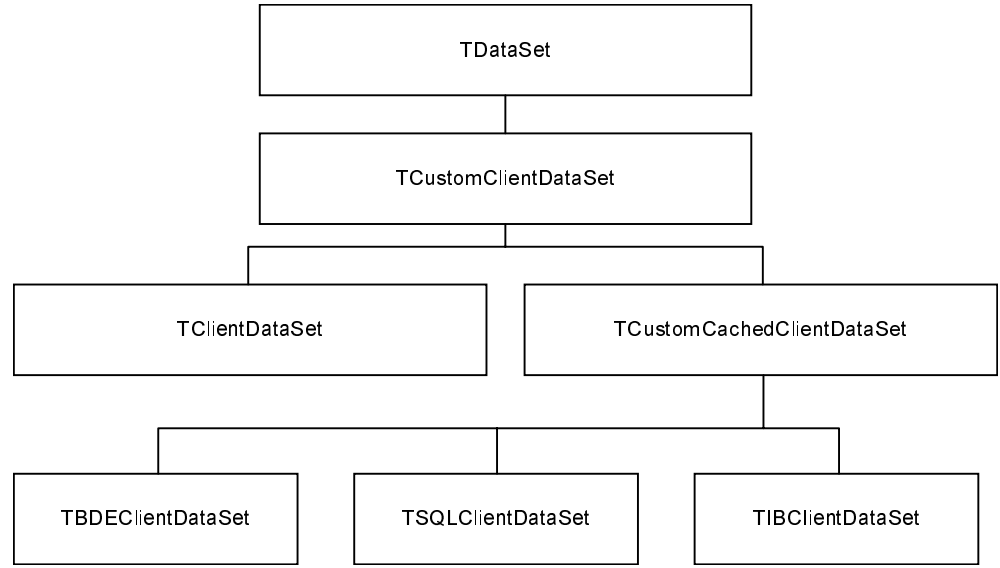


Рис. 10.2. Иерархия классов клиентских наборов данных

А соединение с источником данных осуществляется не с помощью свойства `RemoteServer` (оно будет рассмотрено ниже применительно к компоненту `TClientDataSet`), задающего удаленный сервер, а стандартными средствами соответствующей технологии доступа к данным.

Таким образом, для работы с удаленными данными (т. е. внешними по отношению к клиенту) пригоден только компонент `TClientDataSet`, умеющий работать с внешним провайдером данных.

10.3. Компонент *TClientDataSet*

Компонент `TClientDataSet` используется в клиентской части многозвенного распределенного приложения. Он содержит набор данных, переданный при помощи компонента-провайдера из удаленного набора данных. Компонент обеспечивает выполнение следующих основных функций:

- получение данных от удаленного сервера и передачу ему с использованием удаленного компонента-провайдера сделанных изменений;
- представление набора данных при помощи локального буфера и поддержку 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 тем же набором функций, что и обычный компонент, содержащий набор данных. Основное отличие заключается в том, что источник данных для него доступен только через удаленный компонент-провайдер. Это означает, что сохранение изменений и обновление набора данных традиционными методами 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 разработчик может легко управлять ими.

Как и обычный компонент, компонент TClientDataSet может использоваться совместно визуальными компонентами отображения данных. Для этого нужен компонент TDataSource.

Рассмотрим основные функции, реализуемые компонентом TClientDataSet.

10.3.1. Получение данных от компонента-провайдера

Компонент TClientDataSet получает доступ к удаленным данным с помощью компонента соединения DataSnap *(см. гл. 8)*. В зависимости от используемой технологии это могут быть компоненты TDCOMConnection, TSocketConnection, TWebConnection или TCorbaConnection.

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 задать один из компонентов-провайдеров, которые доступны на сервере приложений, выбранном в компоненте соединения.

Если провайдер был подключен правильно, свойство, доступное только для чтения

```
property HasAppServer: Boolean;
```

автоматически принимает значение True.

Теперь компонент готов к приему данных. При использовании 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 (таково значение по умолчанию), передаются все записи набора данных. При нулевом значении свойства 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 используется метод

```
function GetNextPacket: Integer;
```

Например, это можно сделать таким образом:

```
procedure TDataModule1.ClientDataSetAfterScroll(DataSet: TDataSet);
begin
 if ClientDataSet.EOF then ClientDataSet.GetNextPacket;
end;
```

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 это свойство можно использовать не только для чтения, но и для записи, формируя пакет 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. Более подробные сведения о методе `AS_ApplyUpdates` см. в табл. 9.1.

Размер буфера `Data` в байтах возвращает свойство

```
property DataSize: Integer;
```

10.3.2. 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 содержит свойство

```
property Delta: OleVariant;
```

Для передачи изменений на сервер используется 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 сохраненных 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 используется 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, хранящийся на сервере. При этом (и во всех других случаях, когда компонент запрашивает обновление текущей записи) вызываются методы-обработчики

```
property BeforeRowRequest: TRemoteEvent;
property AfterRowRequest: TRemoteEvent;
```

Что же делать, если необходимо восстановить удаленную запись? В обычном наборе данных после сохранения это невозможно. В компоненте `TClientDataSet` существует метод

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

который возвращает набор данных к состоянию, существовавшему до последней выполненной операции редактирования, добавления или удаления записи. Если параметр `FollowChange` имеет значение `True`, курсор набора данных будет указывать 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');
```

Используя сведения о состоянии записи, можно управлять видимостью записей в наборе данных.

Свойство

```
property StatusFilter: TUpdateStatusSet;
```

определяет состояние записей, которые будут отображаться в наборе данных. Например:

```
ClientDataSet.StatusFilter := usDeleted;
```

отобразит в наборе данных только удаленные записи (при этом изменения не сохранены на сервере).

10.3.3. 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 стороне клиента, можно, например, модифицировать запрос SQL на сервере. Но для этого в свойстве `Options` соответствующего 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 — при выполнении запросов или хранимых процедур может измениться порядок следования параметров, поэтому обращаться к параметрам лучше по именам. Например, так:

```
Edit1.Text := ClientDataSet.Params.ParamByName('OutputParam').AsString;
```

Для того чтобы получить текущие значения параметров компонента набора данных на сервере, достаточно использовать метод

```
procedure FetchParams;
```

Перед и после получения параметров от провайдера, компонент клиентского набора данных вызывает методы-обработчики событий:

```
property BeforeGetParams: TRemoteEvent;
property AfterGetParams: TRemoteEvent;
```

10.3.4. Использование индексов

Обычно применение индексов — прерогатива сервера БД. Только табличные компоненты Delphi 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 создавать и использовать локальные индексы.

Правильно созданные и применяемые 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 используется метод

```
procedure AddIndex(const Name, Fields: string; Options: TIndexOptions;
const DescFields: string = ''; const CaseInsFields: string = ''; const
GroupingLevel: Integer = 0);
```

Параметр `Name` определяет имя нового 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;
```

- `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.

В индексе 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 указать поля, на сортировку которых не влияет регистр символов.

Параметры `DescFields` и `CaseInsFields` используются вместо параметра `Options`.

Параметр `GroupingLevel` задает уровень группировки полей индекса. *Более подробную информацию об этом см. в разд. 10.4 данной главы.*

Основные свойства компонента, обеспечивающие управление индексами, совпадают с аналогичными свойствами табличных компонентов *(подробные сведения об этом см. в гл. 5)*. Здесь лишь кратко перечислим их.

При работе с компонентом программист имеет возможность управлять индексами.

Созданный индекс подключается к набору данных свойством

```
property IndexName: String;
```

которое должно включать имя индекса.

Кроме того, можно использовать свойство

```
property IndexFieldNames: String;
```

в котором задается произвольное сочетание имен индексированных полей таблицы. Имена полей разделяются точкой с запятой.

Свойства `IndexName` и `IndexFieldNames` нельзя использовать одновременно.

Число полей, используемых в текущем индексе табличного компонента, возвращает свойство

```
property IndexFieldCount: Integer;
```

Свойство

```
property IndexFields: [Index: Integer]: TField;
```

представляет собой индексированный список полей, входящих в текущий индекс.

Параметры созданных индексов хранятся в свойстве

```
property IndexDefs: TIndexDefs;
```

Класс `TIndexDefs` рассматривался подробно *в гл. 5*.

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 соответствии с порядком следования 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);
```

10.3.5. Сохранение набора данных в файле

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 делать пользователю, который внес много изменений и не может сохранить их на сервере?

Ему следует сохранить набор данных клиента в файле на локальном диске, а при любом удобном случае — загрузить обратно и переслать на сервер.

Для сохранения данных (по существу это буфер `Data`) в файле используется метод

```
procedure SaveToFile(const FileName: string = '';
Format: TDataPacketFormat=dfBinary);
```

Причем, если параметр `FileName` пуст, имя файла берется из свойства

```
property FileName: string;
```

Кроме того, 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);
```

- `dfBinary` — бинарный;
- `dfXML` — формат XML;
- `dfXMLUTF8` — формат XML в кодировке UTF8.

Обратная загрузка данных, из файла или потока 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;
```

10.3.6. Работа с данными типа BLOB

Если набор данных сервера содержит большие поля (например, изображения), передача таких 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, используя свойство

```
property FetchOnDemand: Boolean;
```

По умолчанию оно равно `True` и клиентский набор данных "выкачивает" встретившиеся данные BLOB автоматически. 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;
```

Кроме этого, в свойстве `Options` компонента-провайдера `TDataSetProvider` обязательно должно быть установлено значение:

```
poFetchBlobsOnDemand := True;
```

10.3.7. Представление данных в форм ате XML

Набор данных клиента легко можно представить в формате XML. Для этого достаточно использовать свойство

```
property XMLData: OleVariant;
```

которое возвращает в формате XML данные, содержащиеся в буфере `Data` в бинарном виде.

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;
```

10.4. Агрегаты

Наличие локального буфера данных позволяет компоненту `TClientDataSet` реализовать ряд дополнительных функций, использующих агрегатные операции над полями всего набора данных, загруженного в локальный буфер. К агрегатным функциям относятся:

- `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:

- индексированный список объектов, содержащих агрегатные выражения — агрегаты;
- агрегатные поля, обеспечивающие получение новых значений подобно вычисляемым полям, но использующие группирование записей с помощью агрегатных функций.

10.4.1. Объекты-агрегаты

Вычисления агрегатных выражений для всех записей набора данных выполняются с помощью объектов класса TAggregate. Индексированный список этих объектов содержится в свойстве

```
property Aggregates: TAggregates;
```

компонента TClientDataSet. Прямым предком класса TAggregates является класс TCollection, поэтому к нему можно применять 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** во всплывающем меню или щелкнуть на кнопке **Add New** (крайняя левая на панели инструментов) (рис. 10.3).

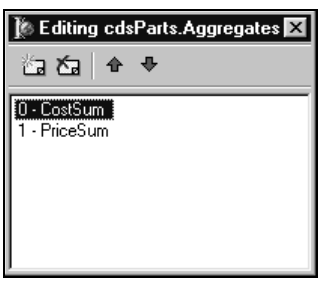


Рис. 10.3. Редактор агрегатов компонента TClientDataSet

Новый агрегат может быть добавлен 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 Америк (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 использовать свойство

```
property InUse: Boolean;
```

Если оно возвращает `True` — вычисляемое выражение агрегата рассчитывается.

Отображением агрегата в визуальных компонентах управляет свойство

```
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 необходимо применять все активные агрегаты, то вместо их последовательного перебора с проверкой свойства `Active` можно использовать свойство

```
property ActiveAggs[Index: Integer] : TList;
```

компонента `TClientDataSet`, которое представляет собой список активных агрегатов.

10.4.2. Агрегатные поля

Агрегатные поля не входят в структуру полей набора данных, так как 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.

Для создания агрегатного поля необходимо использовать команду **New field** из всплывающего меню Редактора полей компонента `TclientDataSet` (рис. 10.4).

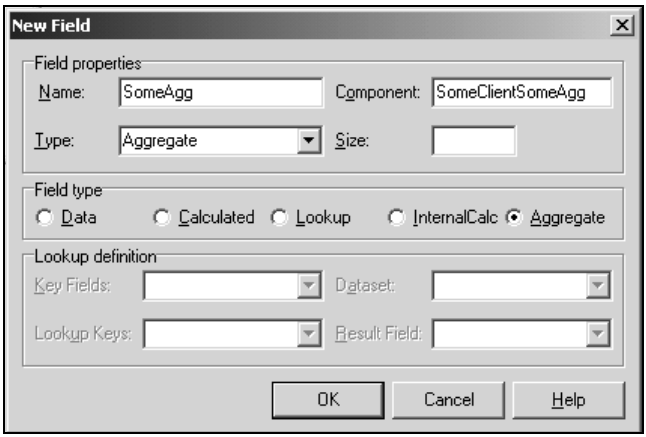


Рис. 10.4. Редактор полей компонента `TClientDataSet`

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 в Редакторе полей они расположены в отдельном списке в нижней части окна (рис. 10.5).

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.

Рис. 10.5. Редактор полей с отдельным списком агрегатных полей

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`, равным `False`.

10.4.3. Группировка полей 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, равном 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.

10.5. Вложенные наборы данных

В *гл. 5* рассматривался вопрос организации между таблицами отношения "один-ко-многим", позволяющего, используя одинаковое значение поля — внешнего ключа, связать одну запись главной таблицы с несколькими записями подчиненной таблицы. Этот широко распространенный в практике программирования приложений БД механизм реализован и в компоненте `TClientDataSet`. Для этого используется класс `TDataSetField`, описывающий поле.

На стороне клиента для создания отношения "один-ко-многим" необходимо применить как минимум два компонента `TClientDataSet`, главный из которых инкапсулирует основной набор данных, а подчиненный — вложенный набор данных.

Итак, на сервере есть два табличных компонента, связанных отношением "один-ко-многим" при помощи свойств `MasterSource` и `MasterFields`. Это могут быть и два компонента запросов SQL, связанные свойством `DataSource` и параметрами подчиненного запроса с одноименными полями главного запроса.

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 всех полей. Для этого достаточно дважды щелкнуть на компоненте и в окне Редактора полей (см. рис. 10.4) 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, доступного только для чтения,

```
property NestedDataSet: TDataSet;
```

Индексированный список всех полей, передаваемых из серверного подчиненного компонента, содержится в доступном только для чтения свойстве

```
property Fields: TFields;
```

В дальнейшем связь между компонентами на стороне клиента 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`, так как их значения утрачивают смысл, 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 с ними записи. При этом вы можете использовать все возможности, предоставляемые компонентом `TClientDataSet` как для основного, так и для вложенного набора данных.

10.6. 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 применяются только в режиме кэширования в обычных компонентах, инкапсулирующих набор данных, и в компоненте `TClientDataSet`. Причем в компоненте `TClientDataSet` реализацию этих свойств обеспечивает локальный кэш.

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 поле, которое было до начала редактирования.

Свойство

```
property NewValue: Variant;
```

содержит новое значение, которое может быть присвоено при обработке ошибки сервера методом-обработчиком `OnReconcileError`.

10.7. Обработка ошибок

Особенности использования 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 использование блоков `try...except` или методов-обработчиков, унаследованных 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 реакцию на ошибку:

```
type TDataAction = (daFail, daAbort, daRetry);
```

- `daFail` — прервать операцию и показать сообщение об ошибке;
- `daAbort` — прервать операцию без сообщения об ошибке;
- `daRetry` — повторить операцию.

Например, для обработки ошибок, возникающих при редактировании набора данных, код обработчика может выглядеть следующим образом:

```
procedure TForm1.ClientDataSetEditError(DataSet: TDataSet; E: EData-
baseError; var Action: TDataAction);
begin
 if Not (DataSet.State in [dsEdit, dsInsert]) then
 begin
  DataSet.Edit;
  Action := daRetry;
 end
 else Action := daAbort;
end;
```

В приведенном примере показано, что если набор данных не находится в состоянии редактирования, то 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, 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);
```

- `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);
```

- `raSkip` — отменить операцию для записей, вызвавших ошибку, с их сохранением в буфере;
- `raAbort` — отменить все изменения для операции, вызвавшей ошибку;
- `raMerge` — совместить (слить) измененные записи с аналогичными записями сервера;
- `raCorrect` — сохранить изменения, сделанные в данном методе-обработчике;
- `raCancel` — отказаться от изменений, вызвавших ошибку, заменив их исходными локальными значениями клиентского набора данных;
- `raRefresh` — отказаться от изменений, вызвавших 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`, имеющий несколько полезных свойств.

Свойство

```
property ErrorCode: DBResult;
```

возвращает код ошибки. Используемые коды ошибок можно найти в файле..\Source\Vcl\DSIntf.pas. Код предыдущей ошибки возвращается свойством

```
property PreviousError: DBResult;
```

Применяя представленную здесь информацию, вы можете самостоятельно управлять обработкой ошибок сервера в клиентском приложении. Но можно поступить проще — использовать стандартное диалоговое окно обработки удаленных ошибок (рис. 10.6). Это окно можно подключить к вашему проекту (оно содержится в модуле ..\ObjRepos\RecError.pas) и вызывать при помощи процедуры

```
function HandleReconcileError(DataSet: TDataSet; UpdateKind: TUpdateKind;
ReconcileError: EReconcileError): TReconcileAction;
```

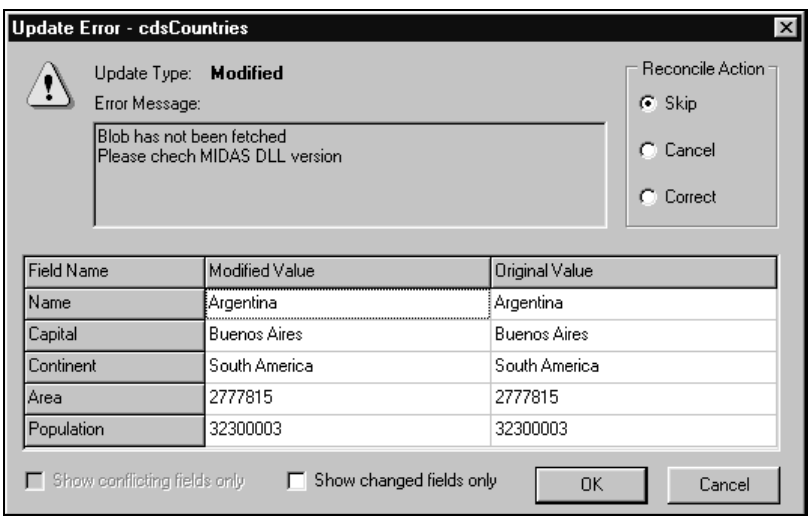


Рис. 10.6. Стандартное диалоговое окно обработки ошибок сервера

Параметрам этой функции присваиваются значения параметров метода-обработчика `OnReconcileError`, а возвращает она действие, выбранное пользователем в диалоговом окне. Таким образом, ее применение очень просто:

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientData-
Set;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAc-
tion);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

10.8. Пример "тонкого" клиента

Пример клиентского приложения является частью группы проектов SimpleRemote.bpg и предназначен для взаимодействия с сервером приложений `SimpleAppSrvr`, процесс создания которого подробно рассматривался в предыдущей главе.

Проект клиента `SimpleClient` (рис. 10.7) 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 на то, что это "обычный" модуль данных (листинг 10.1), используемый в приложениях баз данных *(см. гл. 5)*.
- Главная форма клиентского приложения `fmMain` (файл uMain.pas) содержит визуальные компоненты пользовательского интерфейса.

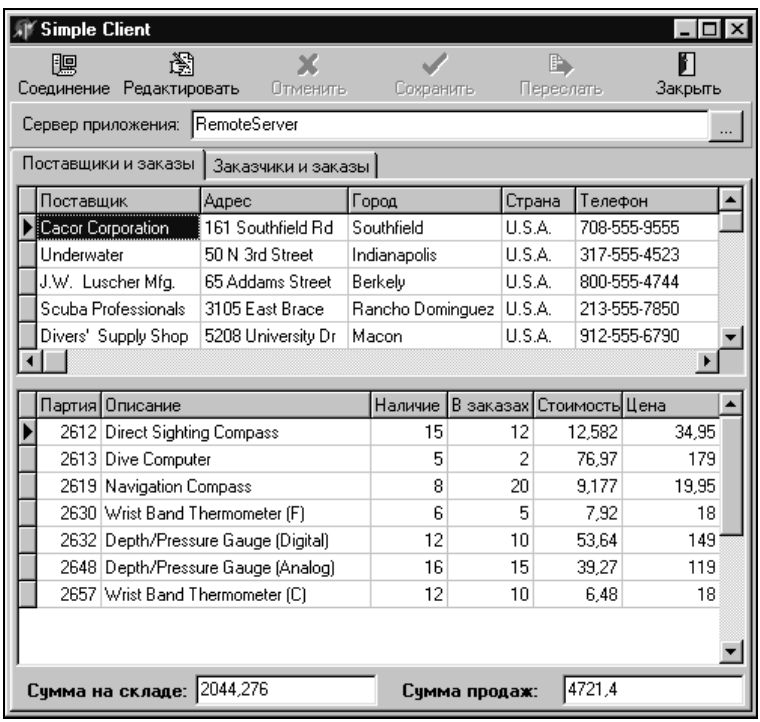


Рис. 10.7. Окно клиентского приложения `SimpleClient`

Листинг 10.1. 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.
```

10.8.1. Соединение клиента с сервером приложения

Для соединения клиентского приложения с сервером в локальной сети используется 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 использовать сервер приложения, установленный на этом же компьютере.

Для настройки соединения компонента `SrvrCon` в свойстве `ComputerName` в Инспекторе объектов должно быть 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 на то, что в этом же списке есть и дочерний модуль данных `Secondary`. Однако для получения доступа к наборам данных дочернего модуля мы не будем создавать еще одно соединение, а воспользуемся компонентом `TSharedConnection`, так как он специально предназначен для подобных случаев. Его настройка выполняется указанием в свойстве `ParentConnection` компонента соединения. В нашем случае это `SrvrCon`.

Для компонента `SrvrCon` предусмотрены два метода-обработчика (см. листинг 10.1) — 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 **Соединение**. При ее нажатии выполняется следующий простой код:

```
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 снова. Специально созданный метод формы `SetCtrlState`

управляет доступностью кнопок формы, анализируя текущее состояние наборов данных.

10.8.2. 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 работников и заказчиков, 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. Поскольку на 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` на этапе разработки 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 10.1).

Поскольку компонент `cdsParts` предназначен и для редактирования данных, то для него необходимо предусмотреть обработку исключительных ситуаций, возникающих не только на стороне клиента, но и на сервере. Для этого используется метод-обработчик `cdsPartsReconcileError` (см. листинг 10.1). Сама операция очень проста и скорее служит демонстрацией возможности создания собственной обработки серверных исключений вместо применения стандартной функции `HandleReconcileError` (см. рис. 10.6). Здесь все изменения в проблемных записях отменяются методом `CancelUpdates`, и выводится сообщение об ошибке.

Локальное редактирование, сохранение или отмена изменений для компонента `cdsParts` выполняется стандартными методами набора данных. Дополнительно при отмене изменений используется метод `UndoLastChange`, позволяющий полностью восстановить первоначальный вариант последней модифицированной записи даже после локального сохранения изменений.

Для передачи изменений серверу использован метод `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 использование при помощи локального буфера. Соединение с удаленным сервером приложений осуществляется компонентами DataSnap.

Часть IV

Основы разработки Internet-приложений

Глава 11. Сокеты

Глава 12. Криптографическая защита информации в Internet

Глава 13. Потоки и процессы

Глава 11

Сокеты

Все распределенные приложения объединяет то, что взаимодействие между их составными частями происходит через некоторую сетевую среду.

Конечно, можно рассматривать эту среду как некий "черный ящик" и не лезть вглубь, в принципы его работы. Тем более, компонентный подход в Delphi способствует этому и позволяет скрыть все детали сетевых соединений.

Несмотря на гигантский прогресс последних лет, надежность сетей не стала стопроцентной, да и вряд ли когда-либо достигнет этого предела. А вы, уважаемый читатель, видимо хотите писать надежные приложения? Тогда вам волей-неволей придется разобраться в принципах работы сетей. Для этого необязательно становиться квалифицированным сетевым инженером; эта и последующие главы дадут вам достаточно знаний, чтобы понять, как и почему транспортируются данные между приложениями.

На каком языке разговаривают между собой элементы компьютерных сетей? В мире общепринятой и стандартизованной является семиуровневая модель структуры протоколов связи под названием *модель взаимодействия открытых систем* (Open Systems Interface, OSI). На каждом из уровней — от первого, физического, до высших уровней представления и приложений — решается свой круг задач и используется свой инструментарий. Мы начнем эту главу с краткого описания модели OSI — без нее дальнейший материал не станет логически связным.

Далее мы перейдем к наиболее распространенному и универсальному API для построения сетевых приложений — спецификации Winsock (Windows Sockets, сокеты Windows). Разумеется, в составе Delphi имеется ряд классов и компонентов, упрощающих работу с сокетами — им мы уделим особое внимание и дадим ряд практических рекомендаций.

Итак, полный перечень тем, рассматриваемых в этой главе:

- введение в теоретические принципы архитектуры сетей;
- применение различных уровней модели OSI в приложениях для Internet;

- сокеты, их назначение и применение;
- иерархия классов Delphi, инкапсулирующих сокеты;
- программная реализация соединения серверного и клиентского сокетов.

11.1. Введение в архитектуру сетей

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.

11.1.1. 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 (для 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

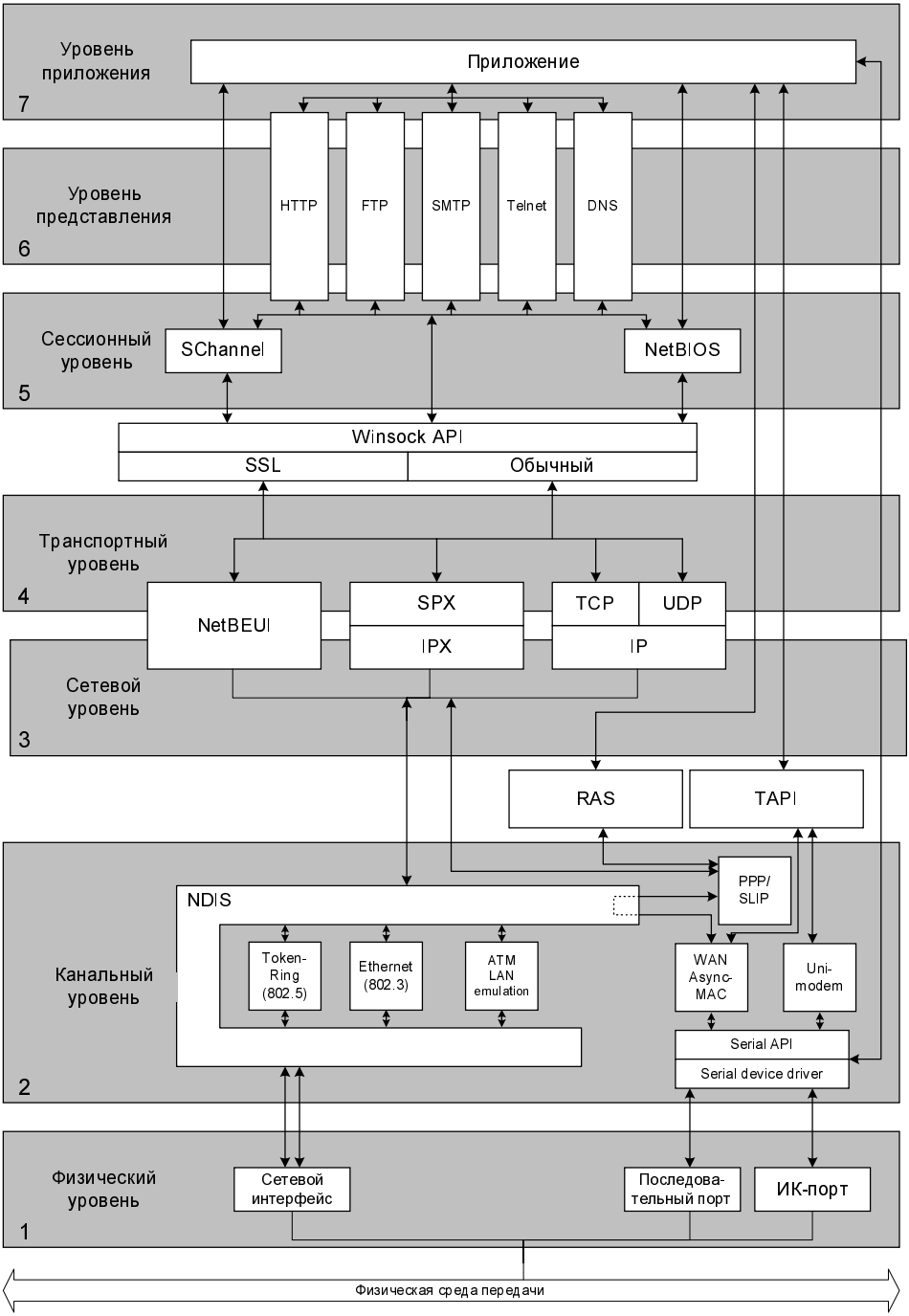


Рис. 11.1. Взаимодействие между уровнями модели OSI

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. 11.1).

Для связи между собой компьютеры должны иметь функционально одинаковые стеки протоколов. Эти стеки обеспечивают связывающимся друг с другом компьютерам возможность общения несмотря на то, что эти компьютеры могут иметь разную архитектуру. Например, компьютеры, использующие процессоры 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 несложного, 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 использует стек 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.

Ниже дается обзор уровней модели OSI. Он вряд ли сделает вас специалистом по сетям, однако поможет понять определенные вещи, необходимые для создания распределенных приложений.

11.1.2. 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 хабом (концентратором) или коммутатором (switch) индивидуальным сетевым кабелем).

Обычно 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.

11.1.3. 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 IEEE (Institute of Electrical and Electronics Engineers, Институтом Инженеров в области электротехники и электроники). 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 9x).

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 побитной 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 прини-

мающему 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 передача, поскольку время приема и передачи данных обеими сторонами синхронизируется. Этого можно достичь двумя путями:

- передача синхронизирующего сигнала в потоке данных;
- использование отдельного канала 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 используется 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 используется 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.

11.1.4. 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 используется 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 использовать 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). Такой 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 выше 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 компенсирован другими протоколами.

Итак, как следует из приведенного выше 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, вы 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 соединения, за которые обычно отвечают сетевые администраторы, зачастую сводятся к неправильной настройке маршрутизации на этом важном уровне.

11.1.5. 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 уставнавливающая 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, использующая UDP 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 Кбайт 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 65535.

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 других; 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.

11.1.6. 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 гл. 12*).

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 HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста). 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.

11.2. Модель клиент/сервер и сокеты

Подробно описанная модель OSI обеспечивает единые правила для любого устройства, которое хочет принять участие в сетевом обмене. Однако эта модель не дает понимания роли каждого участвующего в обмене устройства. Модель клиент/сервер помогает восполнить этот пробел, предлагая сетевому приложению для исполнения две роли, называемые, естественно, клиент и сервер.

Клиентская сторона делает запросы на получение информации и обслуживание. *Серверная сторона* обнаруживает запросы в сети и соответственно отвечает на них в случае получения. В следующих главах мы постараемся научить вас писать приложения, исполняющие обе эти роли.

Связь клиент/сервер работает исходя из предположения, что существует соединение типа "от точки к точке" между клиентом и сервером. К примеру, клиент может располагаться на расстоянии нескольких метров от сервера или даже на другой стороне земного шара. Модель клиент/сервер основана на сетевых концепциях OSI, обеспечивающих виртуальный канал для связи.

На каждом конце этого виртуального канала есть виртуальное оконечное устройство, называемое *сокетом*. Если опять прибегнуть для сравнения к аналоговым коммуникациям и отождествить TCP/IP с телефонной связью, то сокеты — это пара телефонов или факсов, на одном из которых вы набираете номер и ждете соединения, а на втором ваш друг после звонка поднимает трубку и тем самым устанавливает канал связи.

С точки зрения программиста, сокеты — это не более чем удобная и привычная парадигма, представляющая сокеты как "черный ящик", в который можно класть и извлекать данные, не заботясь о том, что у него внутри. Использование сокетов, с одной стороны, позволяет абстрагироваться от частностей работы на нижних уровнях, с другой — решать широкий круг задач, не доступный специализированным протоколам.

Термин "сокет" и сама спецификация этого API впервые появились в ОС UNIX версии BSD (Berkeley Software Distribution) в 80-х, когда происходило становление современной сети Internet и компьютеры с этой ОС составляли становой хребет будущей Сети (отсюда и первоначальное название — сокеты Беркли). Сама идеология канала как некой "трубы" для перекачки данных очень близка UNIX; впрочем, сокеты давно и успешно вписались и в Windows. Создавая сокет, вы получаете его дескриптор (`Handle`); в дальнейшем этот дескриптор может быть использован в таких привычных функциях,

как `ReadFile`/`WriteFile`, наравне с файлами, поименованными каналами (named pipes) и последовательными портами. Взгляните на значки компонентов-сокетов в Delphi: это вилка и розетка. Вы вставляете вилку в нужную розетку, и по вашим приложениям начинает течь "информационный ток" данных.

Сокеты предоставляют действительно кросс-платформенный интерфейс прикладного программирования для разработчиков сетевого программного обеспечения. Они также обеспечивают совместимость команд и данных на двоичном уровне, что позволяет приложениям, написанным с применением API сокетов, использовать любую другую подходящую реализацию, предлагаемую различными поставщиками. По существу, сокеты — это программное обеспечение промежуточного слоя (Middleware) для трансляции вызовов, которые делаются приложениями, на язык инструкций сетевых протоколов.

Итак, во всяком установленном соединении участие принимают два сокета. Сокет, требующий соединения, является клиентом. Сокет, удовлетворяющий это требование, является сервером. Роли клиента и сервера могут оказаться расплывчатыми после соединения двух систем друг с другом, поскольку каждая сторона может выполнять задачи, подразумеваемые другой ролью.

Клиент фактически играет активную роль; он инициирует связь, посылая запросы серверу. Сервер пассивен; он простаивает и ожидает поступления запросов, не делая в это время ничего или почти ничего (фоновые процедуры) до тех пор, пока не придет запрос. Когда появляется запрос, сервер активизируется и отвечает источнику запроса. После удовлетворения запроса сервер возвращается к своей роли пассивного прослушивания сети.

11.2.1. Microsoft Windows и сокеты

Программисты, выросшие на платформах Intel+Microsoft, мало что слышали о "сокетах Беркли", зато поголовно знают термин Winsock. Так называется реализация сокетов, разработанная Microsoft для Windows.

Первые версии Winsock были еще 16-разрядными; текущая версия сокетов Windows (2.0) отличается тем, что она может работать не только поверх протокола TCP/IP, но и поверх других транспортных протоколов (IPX/SPX, NetBEUI); впрочем, мы рассматриваем программирование для Internet, и эта возможность интересует нас лишь постольку поскольку.

API Winsock программно совместим с сокетами Беркли и содержит те же вызовы, что и первоначальная спецификация этих сокетов; вместе с тем фирма Microsoft не была бы сама собой, если бы не добавила целый ряд собственных расширений. Если вы будете анализировать состав API, то легко найдете отличия. Изначальные функции и структуры в названиях не содержат префиксов

и начинаются со строчной буквы, например, `accept`, `connect`, `htonl`, `listen` и т. п. Расширения начинаются с префикса WSA: `WSAAccept`, `WSAConnect`. Базовый набор функций в принципе достаточен для работы, за исключением двух функций — `WSAStartup` и `WSACleanup`, которые должны предварять и завершать работу с Winsock соответственно.

Winsock подразумевает поддержку многопоточности в Windows, когда каждый из процессов содержит один или несколько исполняемых потоков. О режимах работы сокетов и связанных с ними потоках мы поговорим позже.

И в заключение — об альтернативах Winsock, точнее, об их отсутствии. Мы еще будем говорить о том, что для большинства реализаций протоколов высокого уровня фирмы Microsoft есть параллельные решения, созданные независимыми разработчиками. Однако практически никто из них не добирался до собственной реализации сокетов. Во-первых, сокеты весьма тесно интегрированы в операционную систему и сетевую инфраструктуру, а во-вторых, это просто колоссальный труд. Поэтому все компоненты и стеки протоколов, с которыми вы будете иметь дело в Delphi, базируются на Winsock. Borland Kylix опирается на реализацию сокетов, которая включается во все версии Linux.

11.3. Работа на уровне сокетов в Delphi

У программистов, использующих Delphi, имеется огромный выбор средств для работы на уровне сокетов. Почему и откуда?

Во-первых, вы можете работать с этим API "один на один", без промежуточных компонентов и объектов в качестве посредников. Прототипы функций содержатся в файле WINSOCK.PAS, поставляемом с Delphi. Их общее количество не так велико, размер файла всего лишь около 40 Кбайт. Но вот режимы работы сокетов, организация ожидания, обработка ошибок и т. п. не так просты и требуют глубоких знаний о сетях. В общем, если ваша проблема не является чересчур специфичной, прямой смысл обратиться к "во-вторых": компонентам и классам, включающим в себя API сокетов.

11.3.1. Компоненты Delphi, инкапсулирующие сокеты

Исторически первыми появились компоненты `TClientSocket` и `TServerSocket` (модуль `ScktComp`). Они, естественно, были ориентированы на библиотеку WinSock 2. Их объектная модель достаточно проста и логична: несколько свойств, соответствующих начальным установкам сокета, методы, описывающие его функциональные возможности (установка/разрыв соединения, чтение/запись), и события, происходящие в моменты изменения состояния сокета. Дальше мы будем подробно рассматривать эти классы.

Но в Delphi 6 потребовались компоненты, которые можно было бы совместно использовать с библиотекой CLX для Kylix. Модуль же `SocketComp` слишком ориентирован на Windows. По этой причине или по какой-то другой, но фирма Borland предложила другую реализацию оболочки для сокетов — компоненты `TTCPClient`, `TTCPServer` и `TUDPSocket` (модуль `Sockets`).

Еще две библиотеки не принадлежат перу программистов из Borland. Дело в том, что уже несколько лет менеджеры, отвечающие за выпуск Delphi, пытаются снабдить этот продукт альтернативным стеком протоколов, чтобы дать возможность разрабатывать приложения, никак не связанные с Internet-решениями от фирмы Microsoft. Первым таким решением был набор протоколов FastNet (NetMasters LLC, **www.netmastersllc.com**). В Delphi 6 он поддерживается только из соображений совместимости, во всяком случае, версия библиотеки FastNet (5.3.0, build 1055) не изменилась со времени Delphi 5. Явными минусами этой библиотеки является большое количество ошибок и отсутствие исходного кода.

На смену пришла другая реализация набора протоколов — Indy (Internet Direct) от компании Nevrona (**www.nevrona.com**). Этот продукт изначально разрабатывался под названием Internet Component Suite и был доступен в исходных кодах, что позволило устранить множество ошибок.

Вернемся к компонентам `TServerSocket` и `TClientSocket`, имеющимся в Delphi 6 на странице **Internet** Палитры компонентов. Отличия от аналогов из других упомянутых реализаций у них невелики и сводятся в основном к использованию других названий для аналогичных свойств и методов; там, где различия существенны, мы будем специально оговаривать их.

11.3.2. Установление соединения

Механизм соединения при помощи сокетов таков. На одной стороне создается клиентский сокет. Для инициализации связи ему нужно задать путь к серверному сокету, с которым предстоит установить соединение.

Как мы знаем, путь в сетях TCP/IP задается двумя параметрами: адресом или равноценным ему именем хоста (или просто говорят "хостом") и номером порта. Хост в данном случае — это система, в которой запущено приложение, содержащее сокет. Имя хоста, как правило, — символьная строка, поставленная в соответствие адресу и записанная по правилам UNC (Universal Naming Convention), например **http://www.microsoft.com**. Взаимное соответствие между именами и адресами может устанавливаться по-разному, в зависимости от масштаба сети и применяемой ОС. В Internet существует система имен доменов (Domain Naming System, DNS) — протокол, который предназначен для специальных серверов, хранящих и поддерживающих таблицы соответствия между символьным именем и адресом. Мнемоническое имя легче запоминается, но соединение по адресу проще и надежнее, так как не нужно обращаться к DNS за дополнительной информацией.

Примечание

Скорее всего, вы начнете отлаживать и клиентскую, и серверную часть приложения на одной и той же машине. В этом случае можно указать, что сервер находится на том же компьютере и связать сокеты четырьмя способами:

- указанием сетевого имени вашего компьютера (узнать его можно из оснастки **Сеть Панели управления**);
- указанием IP-адреса вашего компьютера (узнать его можно в свойствах протокола TCP/IP; на машине должен быть установлен этот протокол и наличествовать постоянный IP-адрес);
- указанием предопределенного имени localhost, всегда соответствующего данному хосту (т. е. адресат в этом случае совпадает с отправителем);
- Заданием IP-адреса 127.0.0.1 (также всегда указывающего на данный хост).

Номер порта — простое средство для поддержания одновременно нескольких соединений между двумя хостами. Программист, прежде чем установит и использует свое значение, должен ознакомиться со списком уже закрепленных портов.

Итак, задание адреса на клиентской стороне у компонента `TClientSocket` происходит посредством свойств `Address` и `Port`.

С другой вступающей в связь стороны запускается серверный сокет. Первоначально он находится в состоянии *прослушивания* (listening), то есть ожидания связи по заданному порту. После получения запроса от клиента устанавливается связь. Но в то же время создается новый сокет для продолжения прослушивания.

11.3.3. Синхронизация сокетов

Очень важной характеристикой сокетов является их тип — блокирующий (синхронный) или неблокирующий (асинхронный). По существу, работа с сокетами — не что иное, как операции ввода/вывода, которые, как мы знаем, также могут быть синхронными и асинхронными (отложенными). В первом случае при вызове функции ввода/вывода приложение блокируется до ее окончания. Во втором — инициируется ввод/вывод, и выполнение приложения сразу же продолжается; окончание ввода/вывода будет "ознаменовано" возникновением некоторого события в системе. В библиотеке WinSock 2.0 поддерживаются оба типа операций с сокетами; соответственно, в компонентах Delphi также можно установить нужный тип в свойствах `ServerType` и `ClientType`, о которых будет рассказано дальше.

Специфика компонентов `TServerSocket` и `TClientSocket` в том, что они являются "двухэтажными" надстройками над API сокетов. И у того, и у другого имеется свойство:

- `property Socket: TClientWinSocket;` — у компонента `TClientSocket` и
- `property Socket: TServerWinSocket;` — у компонента `TserverSocket`.

Это свойство представляет собой класс — собственно оболочку сокета, со всеми функциями, поддерживающими установку соединения, чтение и запись. Разделение труда между ними таково — на уровне `TServerSocket (TClientSocket)` сосредоточены основные опубликованные (published) свойства и события, обработку которых можно запрограммировать; на уровне `TServerWinSocket (TClientWinSocket)` следует искать методы, в том числе чтения из сокета и записи в него.

Рассмотрим компоненты и классы в такой последовательности: сначала серверную сторону, а затем — клиентскую.

11.3.4. Класс *TServerWinSocket*

На уровне этого класса ведется список соединений с клиентскими сокетами, содержащийся в свойстве:

```
property Connections[Index: Integer]: TCustomWinSocket;
```

Общее число соединений (и число элементов в свойстве `Connections`) равно значению свойства:

```
property ActiveConnections: Integer;
```

Этими списком и счетчиком удобно пользоваться для рассылки всем клиентам какой-нибудь широковещательной информации, например:

```
for i:=0 to ServerSocket.Socket.ActiveConnections-1 do
    ServerSocket.Socket.Connections[i].SendText('Hi!');
```

Тип сервера (блокирующий/неблокирующий) задается свойством

```
type TServerType = (stNonBlocking, stThreadBlocking);
property ServerType: TServerType;
```

Поскольку сервер, который блокируется каждым чтением/записью, представить себе трудно, разработчики фирмы Borland пошли таким путем. Блокирующий режим заменен режимом `stThreadBlocking`. В этом случае при установлении каждого нового соединения запускается отдельный программный поток (объект класса `TServerClientThread`). Он отвечает за связь с отдельным клиентом, и его блокировка не влияет на работу остальных соединений.

Если вы не хотите порождать объект класса `TServerClientThread`, а хотите описать свой класс потока и использовать его для работы с сокетом, вам нужно создать обработчик события:

```
type TGetThreadEvent = procedure (Sender: TObject; ClientSocket:
TServerClientWinSocket; var SocketThread: TServerClientThread) of object;
property OnGetThread: TGetThreadEvent;
```

В отличие от `stThreadBlocking`, тип `stNonBlocking` своим поведением ничем не отличается от рассмотренного выше — операции происходят асинхронно,

и программист должен лишь описать реакцию на события, возникающие в момент окончания операций.

Как известно, создание и уничтожение нового программного потока влечет за собой определенные системные накладные расходы. Чтобы избежать этого, в рассматриваемом классе поддерживается кэш потоков. По завершении соединения потоки не уничтожаются, а переводятся в состояние ожидания нового соединения.

Свойство

```
property ThreadCacheSize: Integer;
```

задает количество свободных потоков, которые могут находиться в готовности для установления соединения с клиентом. Это количество должно рассчитываться в зависимости от интенсивности и продолжительности контакта с клиентами. Лишние потоки поглощают системные ресурсы, в первую очередь память и процессорное время. Чтобы оптимизировать использование кэша свободных потоков, полезно поинтересоваться значением двух свойств:

```
property ActiveThreads: Integer;
property IdleThreads: Integer;
```

показывающих число активных (занятых обслуживанием клиентов) и простаивающих (ожидающих) потоков соответственно.

Старт и завершение потока, работающего с сокетом, обрамлены событиями:

```
type TThreadNotifyEvent = procedure (Sender: TObject;
Thread: TServerClientThread) of object;
property OnThreadStart: TThreadNotifyEvent;
property OnThreadEnd: TThreadNotifyEvent;
```

Конкурирующие за тот или иной системный ресурс потоки могут создавать проблемы при совместном доступе *(см. гл. 13, посвященную многопоточным приложениям)*. Чтобы избежать этого при работе с сокетами, имеются два метода:

```
procedure Lock;
procedure Unlock;
```

Если у вас предусмотрен код, который может вызвать проблемы в многозадачной среде, заключите его между вызовами методов `Lock` и `Unlock` — на это время остальные потоки, работающие с сокетами, будут блокированы.

Методы чтения и записи для блокирующего и неблокирующего режимов существенно отличаются. Рассмотрим сначала те, что предназначены для неблокирующего (асинхронного) режима.

Средства для организации чтения представлены группой из трех методов:

- `function ReceiveLength: Integer;` — возвращает число байтов, которые могут быть приняты в ответ на оповещение клиента о передаче;
- `function ReceiveText: string;` — возвращает прочитанную из сокета текстовую строку;

- `function ReceiveBuf(var Buf; Count: Integer): Integer;` — возвращает данные, прочитанные из сокета в буфер `Buf`, в количестве `Count` байтов.

Аналогично, методы:

```
function SendBuf(var Buf; Count: Integer): Integer;
procedure SendText(const S: string);
function SendStream(AStream: TStream): Boolean;
```

посылают клиенту буфер, текстовую строку и поток данных. В дополнение к этому метод:

```
function SendStreamThenDrop(AStream: TStream): Boolean;
```

посылает клиенту поток данных и завершает соединение.

Примечание

Передаваемый в качестве параметра последних двух функций поток данных `AStream` переходит "под надзор" объекта класса `TServerWinSocket` и удаляется им по мере пересылки. Программист не должен предпринимать попыток удалить `AStream` после вызова методов `SendSrteam` или `SendSrteamThenDrop`.

При помощи метода:

```
function GetClientThread(ClientSocket: TServerClientWinSocket):
TServerClientThread;
```

можно получить указатель на поток, занимающийся обслуживанием конкретного сокета.

События:

```
property OnClientConnect;
property OnClientDisconnect;
property OnClientRead;
property OnClientWrite;
```

имеют одинаковый тип:

```
TSocketNotifyEvent = procedure (Sender: TObject; Socket:
TCustomWinSocket) of object;
```

Они происходят при соединении/отключении от клиента, а также при чтении и записи. Появление ошибки вызывает событие:

```
TErrorEvent = (eeGeneral, eeSend, eeReceive, eeConnect, eeDisconnect, eeAccept);

  TSocketErrorEvent = procedure (Sender: TObject; Socket: TCustomWinSocket;
ErrorEvent: TErrorEvent; var ErrorCode: Integer) of object;
property OnClientError: TSocketErrorEvent;
```

Параметры его имеют следующее назначение. `ErrorEvent` указывает на тип операции, во время которой произошла ошибка. При этом `ErrorCode` содержит код ошибки Windows. Если вы сами обработали ошибку и не хотите дальнейших действий системы по ее обработке, нужно установить параметр `ErrorCode` в 0.

11.3.5. Компонент *TServerSocket*

Самое главное свойство этого компонента — уже упоминавшаяся ссылка на объект:

```
property Socket: TServerWinSocket;
```

Именно благодаря ему доступны все функциональные возможности сокета. Компонент же создан только для того, чтобы опубликовать необходимые свойства и события. В нем имеются свои события `OnClientConnect`, `OnClientDisconnect`, `OnClientRead`, `OnClientWrite`, `OnClientError`, но они не самостоятельны, а только отсылают к соответствующим событиям объекта класса `TServerWinSocket`. Также обстоит дело и со свойствами `ServerType` и `ThreadCacheSize`.

Дополнительно в компоненте предусмотрены события:

- `property OnListen: TSocketNotifyEvent;` — происходит после того, как заданы адрес и порт сокета, и перед тем, как он включается в режим прослушивания (готовности к соединению);
- `property OnAccept: TSocketNotifyEvent;` — происходит непосредственно после установки соединения.

Свойство `property Active: Boolean;` отвечает за состояние сокета. Для клиентского сокета изменение его значения соответствует подключению/отключению от сервера. Для серверного — включение/выключение состояния прослушивания. Использование этого свойства равносильно применению следующих методов:

```
procedure Open;
procedure Close;
```

Свойство

```
property Service: string;
```

пригодится для идентификации назначения сокета. Здесь должно храниться символьное имя сервиса, для которого используется сокет (ftp, http, telnet и др.)

11.3.6. Класс *TClientWinSocket*

Многие из событий и методов этого класса уже описаны *(см. разд. 11.3.4 данной главы)*, так как они имеют общего предка. Но есть и различия, требующие комментария. Как и серверный, клиентский сокет может быть двух типов:

```
type TClientType = (ctNonBlocking, ctBlocking);
property ClientType: TClientType;
```

В отличие от сервера, в блокировке клиента большой беды нет. Если установлен режим `ctBlocking`, клиентское приложение блокируется до завершения операции. Если это вас пугает, отдельный поток для работы с сокетом в режиме `ctBlocking` предлагается создать самому. В режиме `ctNonBlocking` операции выполняются асинхронно.

11.3.7. Компонент *TClientSocket*

Основное внимание при рассмотрении этого компонента обратим на последовательность событий, происходящих при подключении клиентского сокета к серверу. Она такова.

1. Вызывается метод `Open` (или свойство `Active` устанавливается в `True`).
2. Перед началом инициализации происходит событие `property OnLookup: TSocketNotifyEvent;`.
3. В этот момент еще можно поменять свойства объекта типа `TClientWinSocket`: адрес, номер порта и т. п.
4. Сокет полностью инициализируется и начинает поиск. Когда серверный сокет обнаружен, происходит событие

   `property OnConnecting:TSocketNotifyEvent;`
5. Когда клиентский запрос удовлетворен сервером и установлено соединение, происходит событие `property OnConnect: TSocketNotifyEvent;`

Проиллюстрируем сказанное на примере пересылки информации о дате и времени. Процесс подключения к серверу выглядит таким образом:

```
procedure TClientForm.ConnectToServer;
var sAddr: string;
begin
  if ClientSocket.Active then ClientSocket.Active := False;
  try
   if not InputQuery('Select Host', 'Host Address', sAddr) then Abort;
   if Length(sAddr) = 0 then Abort;
   with ClientSocket do
    begin
     Host := sAddr;
     Active := True;
    end;
  except
    TimeSpeedButton.Down := False;
    MemSpeedButton.Down := False;
  end;
end;
```

После установки соединения клиент реагирует на событие `OnClientRead`:

```
procedure TClientForm.ClientSocketRead(Sender: TObject; Socket:
TCustomWinSocket);
  var s: string;
begin
  s:= Socket.ReceiveText;
```

```
  if ((s[1]='T') and (TimeSpeedButton.Down)) then
   TimeSpeedButton.Caption := Copy(s,2,Length(s))
  else if ((s[1]='M') and (MemSpeedButton.Down)) then
   MemSpeedButton.Caption := Copy(s,2,Length(s));
end;
```

В серверном приложении сокет устанавливается в активное состояние (прослушивание) при запуске программы. Все подключившиеся клиенты автоматически заносятся как элементы в список (свойство `Connections`). Саму информацию о дате и времени сервер рассылает по таймеру в виде отформатированных текстовых строк:

```
procedure TServerForm.Timer1Timer(Sender: TObject);
var i: Integer; s : string;
 ms : TMemoryStatus;
begin
 with ServerSocket.Socket do for i:=0 to ActiveConnections-1 do
  Connections[i].SendText('T'+TimeToStr(Now));

  GlobalMemoryStatus(ms);
  s := Format('%10dK',[(ms.dwAvailPageFile + ms.dwAvailPhys) div 1024]);
 with ServerSocket.Socket do for i:=0 to ActiveConnections-1 do
  Connections[i].SendText('M'+s);
end;
```

Полностью текст этого примера приводится на прилагаемом к книге носителе.

Сервер может отреагировать на сообщение от клиента. Ответ следует отправлять через параметр `Socket` произошедшего события `OnClientRead`:

```
procedure TSomeForm.ServerSocketClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add(Socket.ReceiveText);
  Socket.SendText('All right');
end;
```

Резюме

С изучения фундамента — сокетов — начинаем восхождение к освоению протоколов и интерфейсов, используемых для создания современных распределенных приложений и Web-служб. Познакомившись с понятием стека сетевых протоколов и программным интерфейсом сокетов, вы должны быть готовы к тому, чтобы распознать и обработать ошибку, возникшую на любом уровне взаимодействия.

Глава 12

Криптографическая защита информации в Internet

За последнее десятилетие Internet перешел в зрелый возраст и из средства общения энтузиастов и фанатиков стал ареной для ведения серьезного 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 95, 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.

12.1. 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 (защиты от несанкционированного чтения информации) (privacy);
- аутентификация (проверка подлинности) информации и идентификация законных пользователей системы;
- контроль целостности информации.

12.1.1. 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 дешифрования, в результате которого данные будут однозначно восстановлены (рис. 12.1).

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).

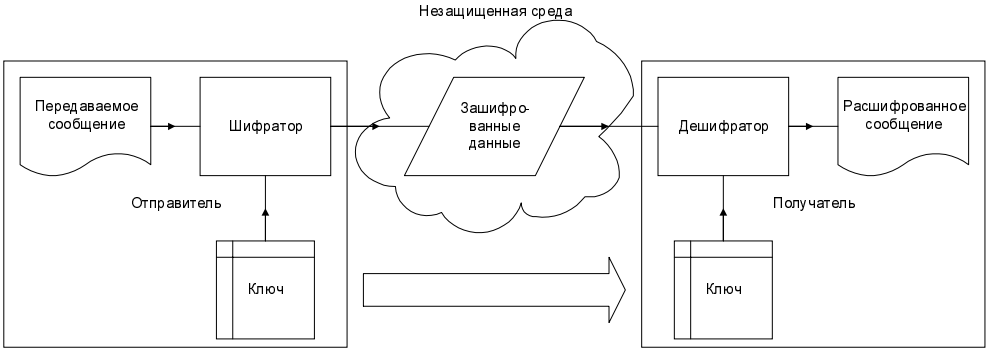


Рис. 12.1. Схема организации доставки секретной информации

К счастью, в цифровой криптографии дело обстоит не так. Есть алгоритмы, в которых ключ шифрования и дешифрования 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 — получает возможность шифровать и отправлять ложную информацию, выдавая себя за подлинного отправителя.

В 1976 году 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 — используется для шифрования. Второй ключ — закрытый, или приватный — применяется 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 веке 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 — 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 применить 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 века стал легкой добычей любителей — энтузиастов криптографии, объединивших для взлома сотни компьютеров по всему миру, работавших параллельно с использованием 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*.

12.1.2. 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.

12.1.3. 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.

12.2. 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*.

До этого мы говорили, что схема с двумя ключами может использовать для шифрования как открытый, так и секретный ключ. 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 не применим — это, скорее, процесс подписи. Таким образом, одни и те же алгоритмы могут использоваться 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.

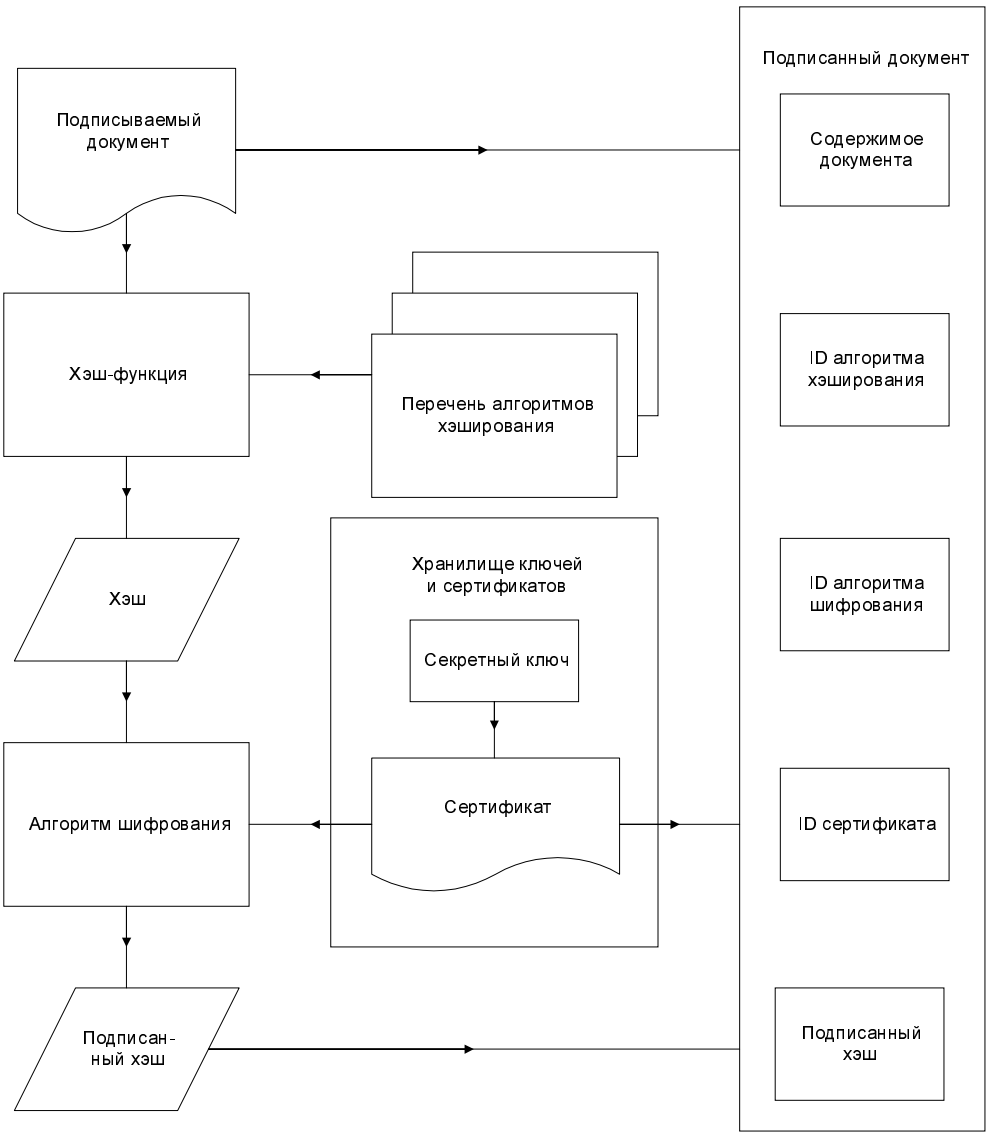


Рис. 12.2. Процедура получения цифровой подписи

Основные этапы создания подписанного документа показаны на рис. 12.2.

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.

На приемной стороне сначала происходит вычисление хэша по тому же самому алгоритму, что и у отправителя. Затем, зная открытый ключ отправителя, 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 — 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 **Contents** диалогового окна **Internet Options** (пункт меню **Tools | Internet Options**). На ней есть кнопка **Certificates**, 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**, **Other People**, **Intermediate Certification Authorities**, **Trusted Root Certification Authorities**. Они соответствуют четырем предустановленным в системе хранилищам сертификатов. На каждой странице — список сертификатов, находящихся в данном хранилище (рис. 12.3).

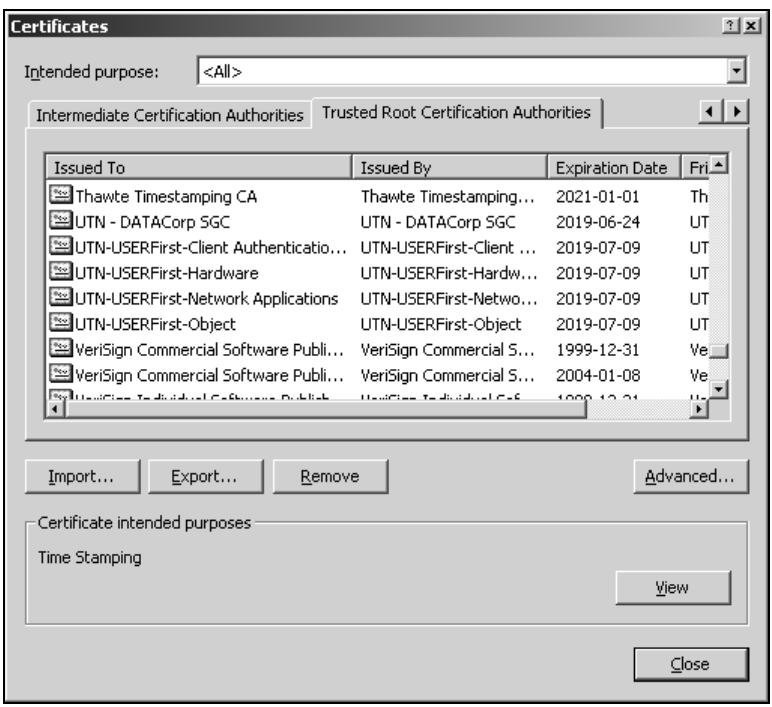


Рис. 12.3. Окно просмотра списка сертификатов в корневом хранилище

Последняя из четырех страниц — **Root** — 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 не обязательно. В Windows XP все драйверы должны быть подписаны.

Сверху от блокнота расположен список выбора "назначение" (**Intended purpose**), 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.

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. 12.4).

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.

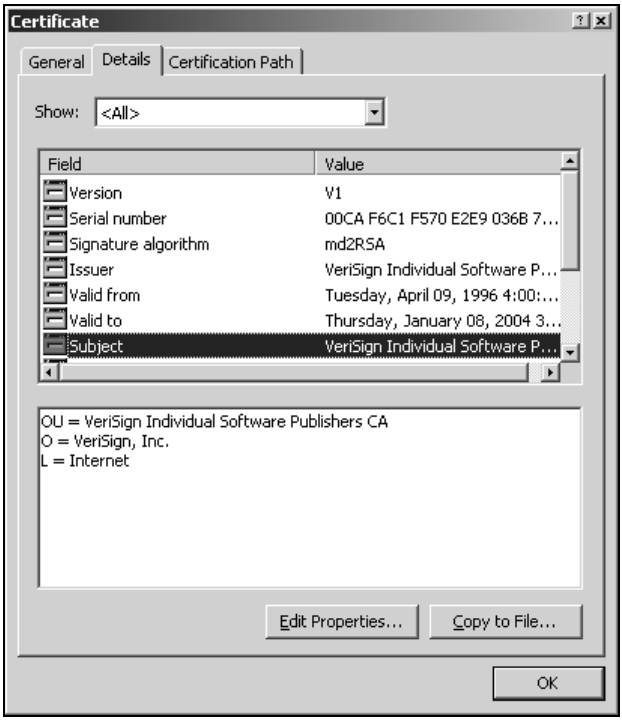


Рис. 12.4. Страница детального просмотра свойств сертификата

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. 12.1); кроме этого, допускаются необязательные расширения (идентификаторы ЦС и владельца сертификата, флаги использования сертификата и т. п.).

Таблица 12.1. *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

Таблица 12.1 *(окончание)*

Название	Содержание
`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.

12.2.1. 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 используется 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 использование и настройка 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 используется 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.

Примечание

Подробную информацию о методах хранения ключей в Windows 2000 можно прочитать в *гл. 13*. Также отметим, что существуют CSP, поддерживающие аппаратное хранение ключей (например, внутри Smart Card).

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 используются 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);
- проверка 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 использования `CryptoAPI` в Delphi необходим файл с описанием его функций. К сожалению, фирма Borland не включает в поставку Delphi трансляцию на Pascal заголовочного файла WinCrypt.h. Для использования `CryptoAPI` 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 несколько неполным.

Наш пример похож на менеджер сертификатов, имеющийся в составе Windows и описанный выше. Тем не менее мы сочли именно такой пример достойным рассмотрения: каковы бы не 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. 12.5).

Наш менеджер работает со стандартными системными хранилищами: **MY** (соответствует странице менеджера сертификатов **Personal**, Личные), **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 функция поддерживается, то число страниц блокнота `TTabControl`, соответствующих хранилищам, увеличивается до их фактического количества.

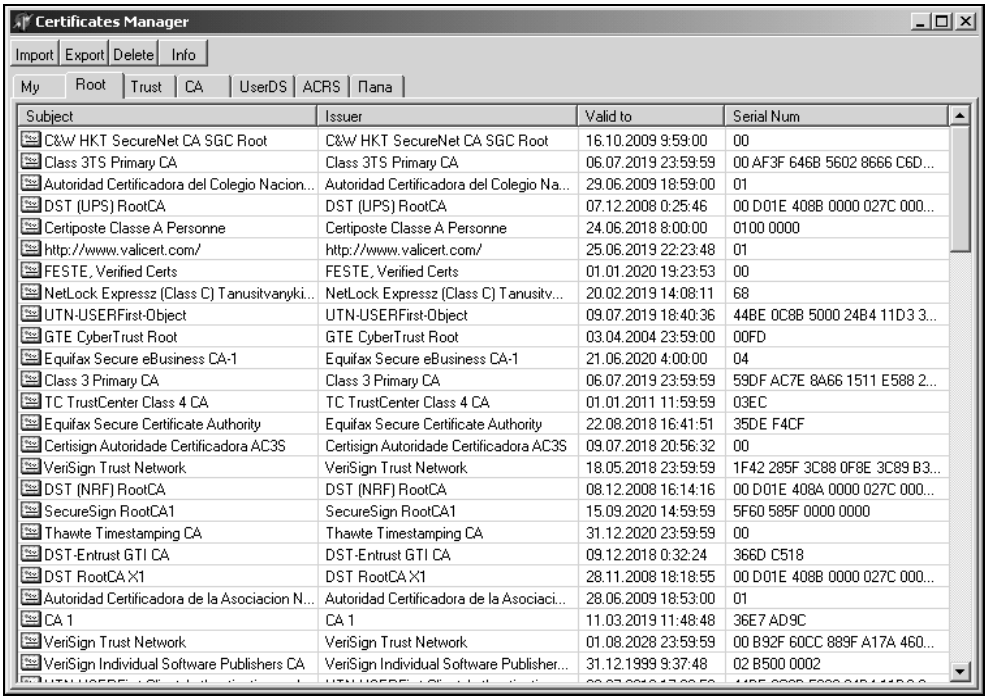


Рис. 12.5. Главное окно Менеджера сертификатов

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**'. Для сертификатов установлены следующие DN-поля (табл. 12.2).

Таблица 12.2. *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
OU	Organizational Unit	Название подразделения, например Certification Services Division
L	Locality	Город, где расположена организация
S	State or Province	Полное название штата или региона

Таблица 12.2 *(окончание)*

Аббревиатура	Полное название	Содержание
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 FILETIME.

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 12.1 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 носителе.) Параметрами функции 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 (см. ниже).

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 12.1. Менеджер сертификатов — функция 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 могут пригодиться вам для использования в других проектах.

12.2.2. 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. 12.6 приведен пример клиентского приложения службы сертификатов, которое использует в качестве интерфейса браузер.

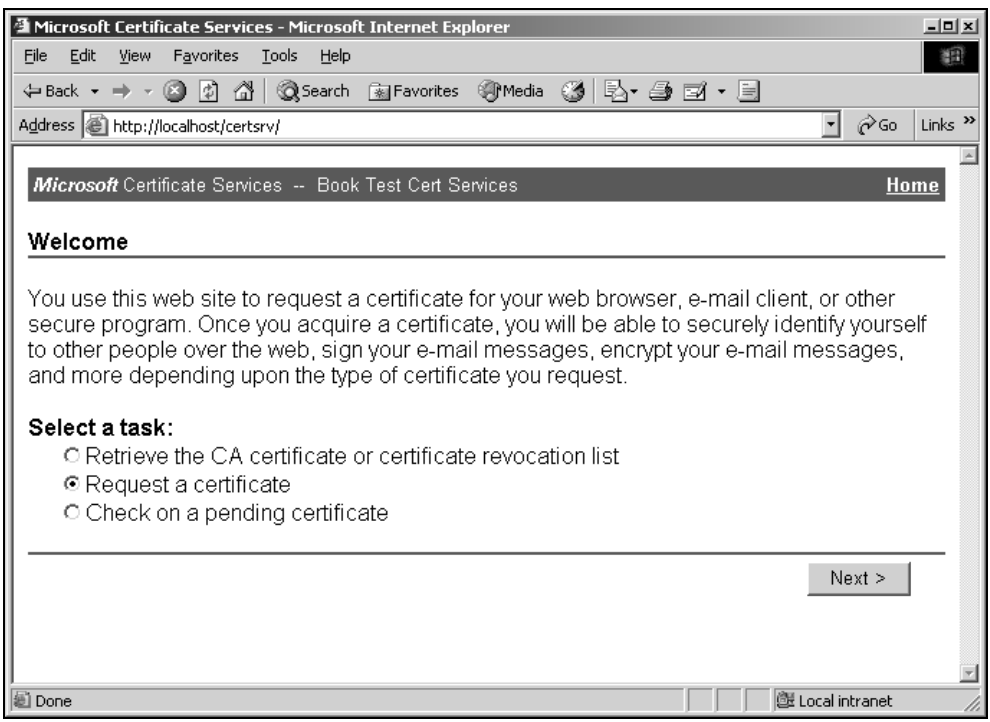


Рис. 12.6. Вид Web-интерфейса для связи со службой сертификатов Microsoft

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 после получения, и, если он удовлетворяет всем условиям, новый сертификат выдается в качестве выходного параметра соответствующего метода.

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 WEB-папка, упомянутая ранее, 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`.

12.3. 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 (Tansport Layer Security) 1.0 с минимальными исправлениями и добавлениями. Фирма Microsoft предпочитает в своих документах ссылаться на термин TLS, но большинство специалистов по-прежнему использует 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; она 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 HTTPS по умолчанию использует 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 этот сертификат.
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** (русский вариант: предупреждать о переключении режимов безопасности), пользователь видит следующее сообщение (рис. 12.7):

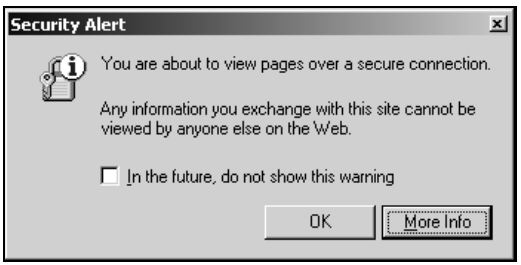


Рис. 12.7. Сообщение Internet Explorer, появляющееся на экране при переключении в режим SSL

Если пользователь IE хочет просмотреть информацию о сертификате сервера, который использовался для соединения по SSL, он может дважды щелкнуть на значке, появляющемся в полосе состояния IE при работе в защищенном режиме (рис. 12.8).

Рис. 12.8. 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 в русской версии). Взведенный флажок заставляет клиента SSL перед установкой соединения запросить у центра выдачи сертификатов свежую версию списка отозванных сертификатов (CRL, certificate revocation list). 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.

12.3.1. Настройка SSL на стороне сервера IIS 5

Рассмотрим настройки защищенных Web-соединений на примере сервера Internet Information Services v.5.0, входящего в состав Windows 2000.

Как следует из описания принципов работы протокола SSL, приведенных раньше, 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 на машинах всех клиентов должен войти в список доверяемых. 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. 12.9). На него большинство пользователей ответит "нет" — и поступит правильно.

Рис. 12.9. Окно соединения по HTTPS с сервером, чей сертификат не является доверяемым

Если вы делаете свое 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. 12.10, *а*);
- в группе элементов управления **Secure Communications** на странице **Directory Security** активна только кнопка **Server Certificate**, открывающая доступ к мастеру установки сертификата (рис. 12.10, *б*).

Не подменяя этой главой документацию по IIS, кратко опишем шаги, необходимые для настройки SSL на сервере.

Сертификат сервера можно выбрать из уже установленных на компьютере подходящих сертификатов (то есть содержащих в поле назначения флаг **Server Authentication**).

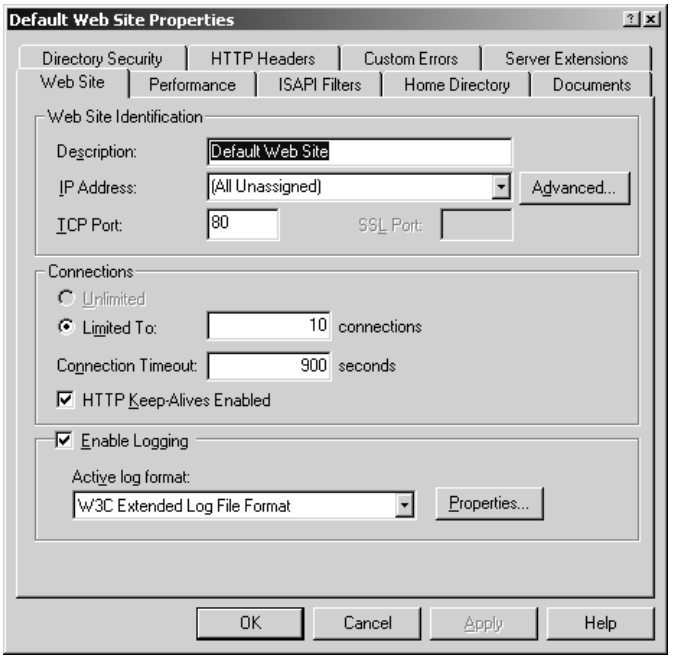
Если вы хотите получить новый сертификат, то запустите специальный мастер создания нового сертификата. На самом деле этот мастер — просто средство для создания запроса в формате PKCS #10 с заранее прописанным назначением — аутентификация сервера. Далее этот запрос направляется в выбранный вами ЦС, и 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**) 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 использовать имя компьютера-сервера; в обоих случаях допустим также IP-адрес.

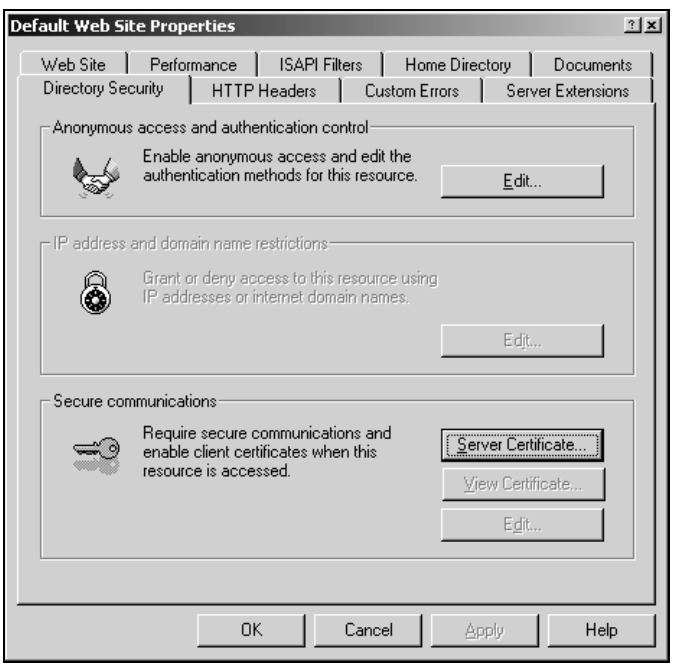
Есть и третий способ получения сертификата — импорт из файла *.key; в этом формате экспортировались сертификаты и соответствующие им секретные ключи в IIS версии 4 и более ранних.

Страница **Directory Security** и соответственно настройки SSL имеются в свойствах каждой папки Web-сайта IIS, но кнопка **Server Certificate** 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** (рис. 12.11).



а



б

Рис. 12.10. Страницы свойств Web-сайта, позволяющие настроить защищенное соединение

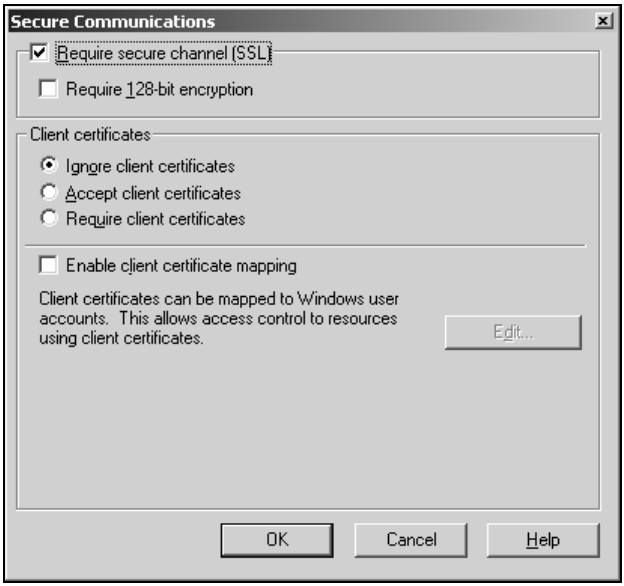


Рис. 12.11. Настройки SSL для Web-папки в Internet Information Services 5

Почти все настройки мы уже рассмотрели; за исключением единственного специфичного для Windows флажка **Enable client certificate mapping**, позволяющего отображать сертификаты, которыми подписались клиенты, на учетные записи пользователей Windows для упрощения управления ресурсами.

12.3.2. 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 6 базируется на библиотеке WinInet, которая представляет собой не 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. Допустим, вы запустили собственный СЦ 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. 12.11). 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. 12.12).

Рис. 12.12. Диалоговое окно запроса клиентского сертификата для соединения по SSL

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.

Альтернативная реализация протокола HTTP есть в пакете компонентов Indy (**www.nevrona.com/Indy**), включенном в состав Delphi 6. При установке защищенного соединения он в свою очередь опирается на пакет OpenSSL (**www.openssl.org**). Оба эти решения поставляются в исходных кодах, что невозможно переоценить, когда речь идет о безопасности. К тому же, Indy — это кросс-платформенный продукт; именно на его основе строятся все Internet-решения для Kylix. Есть возможность перекомпилировать исходные модули VCL для Delphi таким образом, чтобы заменить WinInet на Indy.

Пакет OpenSSL поддерживает — как и средства Microsoft — протоколы SSL версий 2 и 3 и TLS 1.0. Он постоянно обновляется; в момент написания этой книги была доступна версия 0.9.6.

Примечание

Решения на основе SSL подпадают под правила ограничения экспорта в США (U.S. Export Restrictions), поэтому, например, обновления SSL для Indy находятся на сервере в Словении (**www.intellicom.si**). Перед тем как применять и распространять криптографические решения, обязательно ознакомьтесь с правовыми аспектами этой проблемы.

Если вы устанавливаете связь по HTTPS не интерактивно из браузера, а с использованием протокола 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 с использованием 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 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.

Глава 13

Потоки и процессы

Работая с Delphi, нужно иметь в виду: этот замечательный продукт не только упрощает разработку сложных приложений, он использует при этом все возможности операционной системы. Одна из возможностей, которую поддерживает Delphi, — это так называемые *потоки* или нити (threads).

Потоки позволяют в рамках одной программы решать несколько задач одновременно. С недавних пор операционные системы для персональных компьютеров сделали это возможным.

Операционная система (ОС) предоставляет приложению некоторый интервал времени центрального процессора (ЦП), и в момент, когда приложение переходит к ожиданию сообщений или освобождает процессор, операционная система передает управление другой задаче. Теперь, когда компьютеры с несколькими процессорами резко упали в цене, а операционная система Windows NT может использовать их, пользователи получили реальную возможность запускать одновременно более одной задачи. Планируя время центрального процессора, Windows 95 или Windows NT распределяют его между потоками, а не между приложениями. Чтобы использовать все преимущества современных операционных систем, обеспечиваемые несколькими процессорами, программист должен знать, как создавать потоки.

В этой главе рассматриваются следующие вопросы:

- что такое потоки;
- разница между потоком и процессом;
- преимущества потоков;
- использование класса `TThread` в Delphi;
- реализация многопоточного приложения;
- синхронизация потоков.

13.1. Обзор потоков

Определение потока довольно простое: потоки — это объекты, получающие время процессора. Время процессора выделяется квантами (quantum, time slice). *Квант времени* — это интервал, имеющийся в распоряжении потока до тех пор, пока процессор не предоставит свои ресурсы в распоряжение другого потока.

Обратите внимание, что кванты выделяются не программам или процессам, а порожденным ими потокам. Каждый процесс имеет хотя бы один (главный) поток, но современные операционные системы, начиная с Windows 95 (для приверженцев Borland — Kylix: и Linux также), позволяют запустить в рамках процесса несколько потоков.

Если вы только начинаете осваивать применение потоков, самый простой пример их использования — приложения из состава Microsoft Office. К примеру, в пакетах Excel и Word могут действовать несколько потоков. Word умеет одновременно корректировать грамматику и печатать, при этом осуществляя ввод данных с клавиатуры и обрабатывая информацию, поступающую от мыши; программа Excel способна выполнять фоновые вычисления и печатать.

Примечание

Узнать число потоков, запущенных приложением, в Windows NT, 2000 и XP можно при помощи утилиты Task Manager (Диспетчер задач). Для этого среди показателей, отображаемых в окне **Processes**, нужно установить флажок **Thread Count**. Так, в момент набора этих строк MS Word использовал 5 потоков, среда Delphi — 3.

Вполне возможно, что эту главу сейчас вы читаете из чистого любопытства. Но более вероятно, вы обратились к ней в надежде найти ответы на конкретные вопросы. Какого же рода проблемы могут быть решены с применением потоков?

Как правило, алгоритм работы приложения состоит из различных задач: обработки событий, ввода/вывода, связи и других, при этом потоки могут быть органично встроены в программное решение. Если разработчик может разделить большую задачу на несколько мелких, это только повысит переносимость кода и возможности его многократного использования.

Сделав приложение многопоточным, программист получает дополнительные возможности управления им, например, благодаря использованию приоритетов потоков. Если один из них "притормаживает" приложение, занимая слишком много процессорного времени, его приоритет может быть понижен.

Другое важное преимущество внедрения потоков — при возрастании "нагрузки" на приложение можно увеличить их количество и тем самым снять проблему.

Потоки упрощают жизнь тем программистам, которые разрабатывают приложения в архитектуре клиент/сервер. Когда требуется обслуживание нового клиента, сервер может запустить специально для этого отдельный поток. Такие потоки принято называть *симметричными* (symmetric threads) — они имеют одинаковое назначение, исполняют один и тот же код и способны разделять одни и те же ресурсы. Более того, приложения, рассчитанные на серьезную нагрузку, могут поддерживать пул (pool) однотипных потоков. Поскольку создание потока требует определенного времени, для ускорения работы желательно заранее иметь нужное число готовых потоков и активизировать их по мере подключения очередного клиента.

Примечание

Такой подход особенно характерен для Web-сервера Microsoft Internet Information Services и приложений, обрабатывающих запросы в его среде. Если вы создаете приложения ISAPI на Delphi, то вы можете использовать пулинг потоков, подключив к проекту модуль ISAPIThreadPool.pas. Если вы хотите позаимствовать идеи этой технологии для других целей, познакомьтесь с содержимым этого модуля.

Асимметричные потоки (asymmetric threads) — это потоки, решающие различные задачи и, как правило, не разделяющие совместные ресурсы. Необходимость в асимметричных потоках возникает:

- когда в программе выполняются длительные вычисления, при этом необходимо сохранить нормальную реакцию на ввод;
- когда нужно обрабатывать асинхронный ввод/вывод с использованием различных устройств (COM-порта, звуковой карты, принтера и т. п.);
- когда вы хотите создать несколько окон и одновременно обрабатывать в них ввод данных с клавиатуры и от мыши.

13.1.1. Потоки и процессы

Когда мы говорим "программа" (application), то обычно имеем в виду понятие, в терминологии операционной системы обозначаемое как "процесс". Процесс состоит из виртуальной памяти, исполняемого кода, потоков и данных. У него может быть много потоков, но любой процесс обязательно содержит, по крайней мере, один. Поток, как правило, имеет "в собственности" минимум ресурсов; он зависит от процесса, который распоряжается виртуальной памятью, кодом, данными, файлами и другими ресурсами ОС.

Почему мы используем потоки вместо процессов, хотя, при необходимости, приложение может состоять и из нескольких процессов? Дело в том, что переключение между процессами — значительно более трудоемкая операция, чем переключение между потоками. Другой довод в пользу использования потоков — то, что они специально задуманы для разделения ресурсов;

разделить ресурсы между процессами (имеющими раздельное адресное пространство) не так-то просто.

13.1.2. Фоновые процедуры, или как обойтись без потоков

Здесь мы рассмотрим возможность организации фоновых действий внутри однопоточной программы с сохранением реакции этого потока на события от мыши и клавиатуры.

Еще не столь давно программисты пытались эмулировать потоки, запуская процедуры внутри цикла обработки сообщений Windows. Цикл обработки сообщений, или цикл ожидания — это особый фрагмент кода в программе, управляемой событиями. Он исполняется тогда, когда программа находит в очереди события, которые нужно обработать; если таковых нет, программа может выполнить в это время "фоновую процедуру". Такой способ имитации потоков весьма сложен, так как вынуждает программиста, во-первых, сохранять состояние фоновой процедуры между ее вызовами, а во-вторых, определять момент, когда она вернет управление обработчику событий. Если такая процедура выполняется долго, то у пользователя может сложиться впечатление, что приложение перестало реагировать на внешние события. Использование потоков снимает проблему переключения контекста, теперь контекст (стек и регистры) сохраняет операционная система.

В Delphi возможность создания фоновой процедуры реализована с помощью события `OnIdle` объекта `Application`:

```
type TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;
property OnIdle: TIdleEvent;
```

Обработчик этого события вы можете написать, поместив на форму компонент `TApplicationEvents` со страницы **Additional** Палитры компонентов.

Для того чтобы сделать в фоновом режиме какую-то работу, следует разбить ее на кванты и выполнять по одному кванту в каждом вызове `OnIdle` — иначе приложение будет плохо реагировать на внешние воздействия.

13.1.3. Приоритеты потоков

Интерфейс Win 32 API позволяет программисту управлять распределением времени между потоками; это распространяется и на приложения, разработанные в Delphi. Операционная система планирует время процессора в соответствии с приоритетами потоков.

Приоритет потока — величина, складывающаяся из двух составных частей: приоритета породившего поток процесса и собственно приоритета потока. Когда поток создается, ему назначается приоритет, соответствующий

приоритету породившего его процесса. В свою очередь, процессы могут иметь следующие классы приоритетов:

- `Real time` (реального времени);
- `High` (высокий);
- `Above normal` (выше нормального);
- `Normal` (нормальный);
- `Below normal` (ниже нормального);
- `Idle` (фоновый).

Примечание

Классы приоритетов Above normal и Below normal появились впервые в Windows 2000.

Класс реального времени задает приоритет даже больший, чем у многих процессов операционной системы. Такой приоритет нужен для обработки высокоскоростных потоков данных. Если подобный процесс не завершится за короткий промежуток времени, пользователь почувствует, что система перестала откликаться, так как даже обработка событий мыши не получит времени процессора.

Использование класса `High` ограничено процессами, которые должны завершаться за короткое время, чтобы не вызвать сбойной ситуации. Пример — процесс, который посылает сигналы внешнему устройству; причем устройство отключается, если не получает сигнал своевременно. Если у вас возникли проблемы с производительностью вашего приложения, было бы неправильно решать их просто за счет повышения его приоритета до класса `high` — такой процесс влияет на всю ОС. Возможно, в этом случае следует модернизировать ваш компьютер.

Большинство процессов запускается с приоритетом класса `normal`. Нормальный приоритет означает, что процесс не требует какого-либо специального внимания со стороны операционной системы.

И наконец, процессы с фоновым приоритетом запускаются лишь в том случае, если в очереди Диспетчера задач нет других процессов. Обычные виды приложений, использующие такой приоритет, — это программы сохранения экрана и системные агенты (system agents). Программистам следует применять фоновые процессы для организации завершающих операций и реорганизации данных. Примерами могут служить сохранение документа или резервное копирование базы данных.

Приоритеты имеют значения от 0 до 31. Процесс, породивший поток, может впоследствии изменить его приоритет; в этой ситуации у программиста есть возможность управлять скоростью отклика каждого потока.

Базовый приоритет нити складывается из двух составляющих, однако это не означает, что он просто равен их сумме. Взгляните на соответствующие величины, показанные в табл. 13.1. Для потока, имеющего собственный приоритет THREAD_PRIORITY_IDLE, базовый приоритет будет равен 1, невзирая на приоритет породившего его процесса.

Кроме того, для класса Normal приведены два приоритета, снабженные буквами B (Background, передний план) и F (Foreground, фон) (см. табл. 13.1).

Таблица 13.1. *Классы процессов и приоритеты их потоков (для Windows 2000 и XP)*

Класс процесса/ приоритет потока	IDLE_ PRIORITY_ CLASS	BELOW_ NORMAL_ PRIORITY_ CLASS	NORMAL_ PRIORITY_ CLASS	ABOVE_ NORMAL_ PRIORITY_ CLASS	HIGH_ PRIORITY_ CLASS	REALTIME_ PRIORITY_ CLASS
THREAD_ PRIORITY_ IDLE	1	1	1	1	1	16
THREAD_ PRIORITY_ LOWEST	2	4	5 (B) 7 (F)	8	11	22
THREAD_ PRIORITY_ BELOW_ NORMAL	3	5	6 (B) 8 (F)	9	12	23
THREAD_ PRIORITY_ NORMAL	4	6	7 (B) 9 (F)	10	13	24
THREAD_ PRIORITY_ ABOVE_ NORMAL	5	7	8 (B) 10 (F)	11	14	25
THREAD_ PRIORITY_ HIGHEST	6	8	9 (B) 11(F)	12	15	26
THREAD_ PRIORITY_ TIME_ CRITICAL	15	15	15	15	15	31

Помимо базового приоритета, описываемого в табл. 13.1, планировщик заданий (scheduler) может назначать так называемые динамические приоритеты.

У процессов класса NORMAL_PRIORITY_CLASS при переключении их из фонового режима в режим переднего плана и в ряде других случаев приоритет потока, с которым связано окно переднего плана, повышается. Так работают все клиентские операционные системы фирмы Microsoft. Серверные операционные системы оптимизированы для выполнения фоновых приложений. Впрочем, Windows NT и более поздние ОС на этом ядре позволяют переключать режим оптимизации, используя переключатель группы **Application response** апплета **System** Панели управления Windows (рис. 13.1).

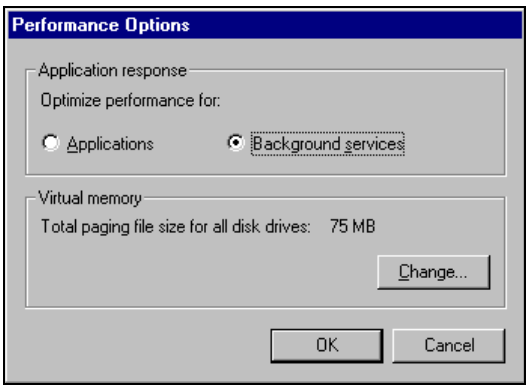


Рис. 13.1. Диалоговое окно **Performance Options**, позволяющее управлять алгоритмом назначения приоритетов

Windows 2000 Professional и Windows 2000 Server имеют разные настройки алгоритма выделения квантов времени. Первая — клиентская — операционная система выделяет время короткими квантами переменной длины, для ускорения реакции приложений переднего плана (foreground). Для сервера же более важна стабильная работа системных служб, поэтому во второй ОС система распределяет длинные кванты постоянной длины.

Теперь нужно обязательно сказать о том, как же планировщик заданий использует приоритеты потоков для распределения процессорного времени.

Операционная система имеет различные очереди готовых к выполнению потоков — отдельную для каждого уровня приоритета. В момент распределения нового кванта времени она просматривает очереди — от высшего приоритета к низшему. Готовый к выполнению поток, стоящий первым в очереди, получает этот квант и перемещается в хвост очереди. Поток будет исполняться на протяжении всего кванта, если не произойдет одно из двух событий:

- выполняющийся поток остановился для ожидания;
- появился готовый к выполнению поток с более высоким приоритетом.

Теперь, наверное, для вас более ясна опасность, исходящая от неоправданного завышения приоритетов. Ведь если есть активные потоки с высоким

приоритетом, ни один поток с более низким приоритетом ни разу не получит времени процессора! Такая проблема может возникнуть даже на уровне вашего приложения. Предположим, вы назначили вычислительному потоку приоритет `THREAD_PRIORITY_ABOVE_NORMAL`, а потоку, в котором обрабатывается ввод пользователя — `THREAD_PRIORITY_BELOW_NORMAL`. Тогда вместо запланированного результата — совместить вычисления с нормальной реакцией приложения — вы получите прямо противоположный. Приложение вообще перестанет откликаться на ввод, и снять его можно будет только с помощью средств ОС.

Итак, нормальная практика для асимметричных потоков — это назначение потоку, обрабатывающему ввод, более высокого приоритета, а всем остальным — более низкого или даже приоритета класса `Idle`, если этот поток должен выполняться только во время простоя системы.

13.2. Класс *TThread*

Delphi предоставляет программисту полный доступ к возможностям программирования интерфейса Win32. Для чего же тогда фирма Borland разработала специальный класс для организации потоков? Вообще говоря, программист не обязан разбираться во всех тонкостях механизмов, предлагаемых операционной системой. Класс должен включать в себя и упрощать программный интерфейс; класс `TThread` — прекрасный пример предоставления разработчику простого доступа к программированию потоков. Сам API потоков, вообще говоря, не очень сложен, но предоставленные `TThread` возможности замечательно просты. В двух словах, все, что вам необходимо сделать, — это перекрыть виртуальный метод `Execute`.

Другая отличительная черта класса `TThread` — это гарантия безопасной работы с библиотекой визуальных компонентов VCL. Без использования класса `TThread` во время вызовов VCL могут возникнуть ситуации, требующие специальной синхронизации *(см. разд. 13.4 данной главы)*.

Нужно отдавать себе отчет в том, что, с точки зрения операционной системы, поток — это системный объект. При создании он получает дескриптор и отслеживается ОС. Объект класса `TThread` — это конструкция Delphi, соответствующая потоку ОС. Этот объект VCL создается до реального возникновения потока в системе и уничтожается после его исчезновения.

Изучение класса `TThread` начнем с метода

```
Procedure Execute; virtual; abstract;
```

Это и есть код, исполняемый в создаваемом вами потоке `TThread`.

Примечание

Хотя формальное описание метода `Execute` — abstract, но мастер создания нового объекта `TThread` создает для вас пустой шаблон этого метода.

Переопределяя метод Execute, мы можем тем самым закладывать в новый потоковый класс то, что будет выполняться при его запуске. Если поток был создан с аргументом CreateSuspended (см. описание конструктора ниже), равным False, то метод Execute выполняется немедленно, в противном случае Execute выполняется после вызова метода Resume.

Если поток рассчитан на однократное выполнение каких-либо действий, то никакого специального кода завершения внутри Execute писать не надо.

Если же в потоке будет выполняться какой-то цикл и поток должен завершиться вместе с приложением, то условия окончания цикла должны быть примерно такими:

```
procedure TMyThread.Execute;
begin
repeat
 DoSomething;
Until CancelCondition or Terminated;
end;
```

Здесь CancelCondition — ваше личное условие завершения потока (данные исчерпаны, вычисления закончены, на вход поступил тот или иной символ и т. п.), а свойство Terminated говорит о завершении потока (это свойство может быть установлено как внутри потока, так и извне; скорее всего, завершается породивший его процесс).

Конструктор

```
constructor Create(CreateSuspended: Boolean);
```

получает параметр CreateSuspended. Если его значение равно True, вновь созданный поток не начинает выполняться до тех пор, пока не будет сделан вызов метода Resume. В противном случае, если CreateSuspended имеет значение False, конструктор завершается, и затем поток начинает исполнение.

Деструктор

```
destructor Destroy; override;
```

вызывается, когда необходимость в созданном потоке отпадает. Деструктор завершает его и высвобождает все ресурсы, связанные с объектом класса TThread.

Для окончательного завершения потока (без последующего запуска) существует метод Terminate

```
function Terminate: Integer;
```

Но если вы думаете, что этот метод выполняет какие-то принудительные действия по остановке потока, вы ошибаетесь. Происходит всего лишь присвоение свойству

```
Property Terminated: Boolean;
```

значения `True`. Таким образом, `Terminate` — это указание потоку завершиться, выраженное "в мягкой форме", с возможностью корректно освободить ресурсы. Если вам нужно немедленно завершить поток, используйте функцию Windows API `TerminateThread`.

Примечание

Метод `Terminate` автоматически вызывается и из деструктора объекта. Поток-объект VCL будет дожидаться, пока завершится поток-объект операционной системы. Таким образом, если поток не умеет завершаться корректно, вызов деструктора потенциально может привести к зависанию всей программы.

Есть еще одно полезное свойство

```
property FreeOnTerminate: Boolean;
```

Если это свойство равно True, то деструктор потока будет вызван автоматически сразу после его завершения. Это очень удобно для тех случаев, когда в вашей программе неизвестен точный момент завершения потока и вы хотите использовать поток по принципу "выстрелил и забыл" (fire and forget).

Метод

```
Function WaitFor: Integer;
```

предназначен для синхронизации и позволяет одному потоку дождаться момента завершения другого потока. Если вы внутри потока `FirstThread` пишете код:

```
Code := SecondThread.WaitFor;
```

то это означает, что поток `FirstThread` останавливается до момента завершения потока `SecondThread`. Метод `WaitFor` возвращает код завершения ожидаемого потока (см. свойство `ReturnValue`).

Свойства:

```
Property Handle: THandle read FHandle;
Property ThreadID: THandle read FThreadID;
```

предоставляют программисту прямой доступ к потоку средствами API Win 32. Если у разработчика возникнет желание обратиться к потоку и управлять им, минуя возможности класса `TThread`, значения `Handle` и `ThreadID` могут быть использованы в качестве аргументов функций Win 32 API. Например, если программист хочет перед продолжением выполнения приложения дождаться завершения сразу нескольких потоков, он должен вызвать функцию API `WaitForMultipleObjects`; для ее вызова необходим массив дескрипторов потоков.

Свойство

```
property Priority: TThreadPriority;
```

позволяет запросить и установить приоритет потоков. Приоритеты потоков подробно описаны выше. Допустимыми значениями приоритета для объектов

класса `TThread` являются `tpIdle`, `tpLowest`, `tpLower`, `tpNormal`, `tpHigher`, `tpHighest` и `tpTimeCritical`.

Метод

```
procedure Synchronize(Method: TThreadMethod);
```

относится к секции `protected`, то есть может быть вызван только из потомков `TThread`. Delphi предоставляет программисту метод `Synchronize` для безопасного вызова методов VCL внутри потоков. Метод `Synchronize` обеспечивает одновременный доступ к каждому объекту VCL только одного потока, что помогает избежать конфликтных ситуаций. Аргумент, передаваемый в метод `Synchronize`, — это имя метода, который обращается к VCL; вызов `Synchronize` с этим параметром равнозначен вызову передаваемого метода. Такой метод (класса `TThreadMethod`) не должен иметь никаких параметров и не может возвращать никаких значений. К примеру, в основной форме приложения нужно предусмотреть процедуру

```
procedure TMainForm.SyncShowMessage;
begin
 ShowMessage(IntToStr(ThreadList1.Count));
//другие обращения к VCL
end;
```

а в потоке для отображения сообщения писать не

```
ShowMessage(IntToStr(ThreadList1.Count));
```

и даже не

```
MainForm.SyncShowMessage;
```

а только так:

```
Synchronize(MainForm.SyncShowMessage);
```

Примечание

Обращаясь к объекту VCL из потока, убедитесь в том, что для этого используется метод `Synchronize`; в противном случае результаты могут оказаться непредсказуемыми. Это верно даже в том случае, если вы применяете средства синхронизации, описанные дальше!

Метод

```
procedure Resume;
```

класса `TThread` вызывается, когда поток возобновляет выполнение после остановки или для явного запуска потока, созданного с параметром конструктора `CreateSuspended`, равным `True`.

Вызов метода

```
procedure Suspend;
```

приостанавливает поток с возможностью его повторного запуска впоследствии, причем не имеет значения, какой код исполнялся потоком в этот момент. Повторное выполнение продолжится с точки останова.

Одноименное свойство

```
property Suspended: Boolean;
```

позволяет программисту определить, не приостановлен ли поток. С помощью этого свойства можно также запускать и останавливать поток. Присвоив `Suspended` значение `True`, вы получите тот же результат, что и при вызове метода `Suspend` — приостановку. Наоборот, если значение свойства `Suspended` равно `False`, возобновляется выполнение потока, как при вызове метода `Resume`.

Свойство

```
Property ReturnValue: Integer;
```

позволяет узнать и установить значение, возвращаемое потоком после его завершения. Эта величина полностью определяется пользователем. По умолчанию поток возвращает ноль, но если программист захочет вернуть другое значение, то простая переустановка свойства `ReturnValue` внутри потока позволит получить эту информацию другим потокам. Это, к примеру, может пригодиться для сообщения о возникших внутри потока проблемах или для возврата количества не прошедших орфографическую проверку слов.

На этом завершим подробный обзор класса `TThread`. Для более близкого знакомства с потоками и классом Delphi `TThread` создадим многопоточное приложение с помощью всего несколько строк кода и щелчков мышью.

13.3. Пример создания многопоточного приложения в Delphi

Этот раздел содержит пошаговое описание алгоритма создания простого, но показательного примера многопоточного приложения. Мы будем пытаться вычислить число "пи" с максимальной точностью. Конечно, встроенная в Delphi константа `Pi` имеет достаточную точность, правильнее сказать — максимальную, обеспечиваемую самым точным 10-байтным форматом для вещественных чисел `Extended`. Так что превзойти ее нам не удастся. Но этот пример использования потоков может послужить прологом для решения реальных задач.

Пример будет содержать два потока: главный (обрабатывающий ввод пользователя) и вычислительный; мы сможем изменять их свойства и наблюдать за реакцией.

Итак, выполните следующую последовательность действий:

1. В среде Delphi в меню **File** выберите пункт **New Application**.
2. Расположите на форме пять меток и один переключатель-флажок, как показано на рис. 13.2.

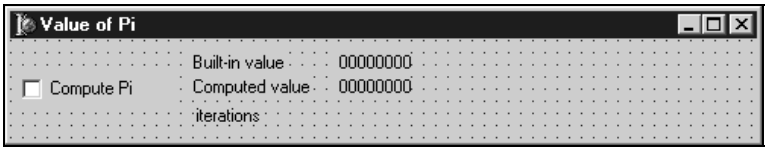


Рис. 13.2. Внешний вид формы приложения ComputationPi

Переименуйте главную форму в `fmMain`.

3. В меню **File** выберите пункт **Save Project As**. Сохраните модуль с именем `uMain`, а проект — `ComputationPi`.
4. В меню **File** выберите пункт **New**. Откроется страница **New Items** Репозитария Delphi, показанная на рис. 13.3. Затем дважды щелкните на объекте типа поток (значок **Thread Object**).

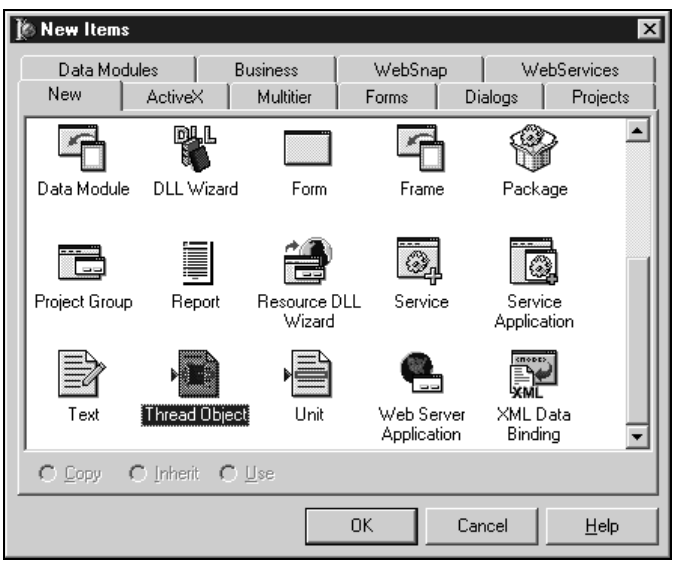


Рис. 13.3. Страница Репозитария **New Items** с выбранным объектом типа "поток"

5. Когда появится диалоговое окно для именования создаваемого потока, введите `TPiThread` и нажмите клавишу <Enter> (рис. 13.4).

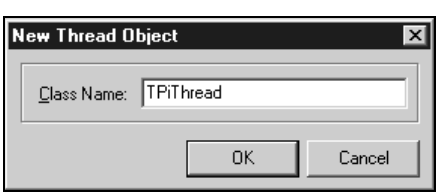


Рис. 13.4. Диалоговое окно **New Thread Object**

Delphi создаст новый модуль и поместит в него шаблон для нового потока.

6. Внесите в метод `Execute` код, который описывает вычисление числа "пи" с использованием сходимости бесконечного ряда Лейбница:

$$\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - \ldots$$

Разумеется, отображать новое значение после каждой итерации — это то же, что стрелять из пушки по воробьям. На отображение информации система потратит в десятки раз больше времени, чем на собственно вычисления. Поэтому мы ввели константу `UpdatePeriod`, которая регулирует периодичность отображения текущего значения.

Код метода `Execute`:

```
const
 //Лучше выбрать нечетное значение во избежание монотонного приближения
 UpdatePeriod = 999999;

procedure TPiThread.Execute;
var sign : Integer;
  PiValue, PrevValue : Extended;
  i : Int64;
begin
  { Place thread code here }
  PiValue := 4;
  sign := -1;
  i := 0;
  repeat
  Inc(i);
  PrevValue := PiValue;
  PiValue := PiValue + sign * 4 / (2*i+1);
  sign := -sign;
  if i mod UpdatePeriod = 0 then
   begin
    GlobalPi := PiValue;
    GlobalCounter := i;
    Synchronize(fmMain.UpdatePi);
   end;
  until Terminated or
  (Abs(PiValue - PrevValue)<1E-19) ;
end;
```

7. В меню **File** выберите пункт **Save As**. Сохраните модуль с потоком как uPiThread.pas.
8. Отредактируйте файл модуля uMain.pas и добавьте модуль `uPiThread` к списку используемых модулей в секции интерфейса. Он должен выглядеть так:

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
   Forms,
  StdCtrls,
  uPiThread;
```

9. В секцию `public` формы `TfmMain` добавьте ссылку на создаваемую нить:

```
PiThread : TPiThread;
```

10. Добавьте в модуль `uPiThread` две глобальные переменные

```
GlobalPi : Extended;
GlobalCounter : Int64;
```

а в модуль uMain — метод `UpdatePi`:

```
procedure TfmMain.UpdatePi;
begin
  if IsIconic(Application.Handle) then Exit;
  LaValue.Caption := FloatToStrF(GlobalPi, ffFixed, 18, 18);
  laIterNum.Caption := IntToStr(GlobalCounter) + ' iterations';
end;
```

Этот метод, если вы обратили внимание, вызывается из потока посредством процедуры `Synhcronize`. Он отображает текущее значение приближения к числу "пи" и количество итераций.

Если главное окно приложения свернуто, то для повышения скорости счета отображение данных не производится. Так что после его развертывания вам, возможно, придется подождать некоторое время для обновления текущих значений переменных.

11. Выполните двойной щелчок на свободном месте рабочей области формы, при этом в тексте модуля uMain.pas появится шаблон метода `FormCreate`. В нем мы опишем отображение значения системной константы Pi:

```
procedure TfmMain.FormCreate(Sender: TObject);
begin
 laBuiltIn.Caption := FloatToStrF(Pi, ffFixed, 18, 18);
end;
```

12. Выберите на форме переключатель-флажок (его название `cbCalculate`) и внесите в его событие `OnClick` код, создающий или уничтожающий вычислительный поток в зависимости от состояния переключателя:

```
procedure TfmMain.cbCalculateClick(Sender: TObject);
begin
 if cbCalculate.Checked then
  begin
   PiThread := TPiThread.Create(True);
   PiThread.FreeOnTerminate := True;
   PiThread.Priority := tpLower;
   PiThread.Resume;
  end
 else
  begin
   if Assigned(PiThread) then PiThread.Terminate;
  end;
end;
```

Теперь многопоточное приложение готово к запуску. Если все пройдет нормально, вы увидите картинку подобную той, которая приведена на рис. 13.5.

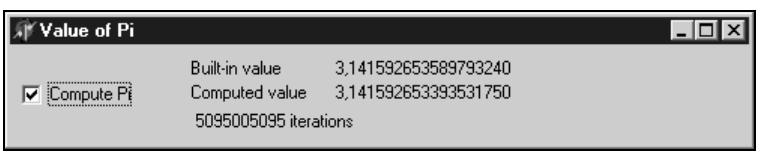


Рис. 13.5. Выполняющееся приложение `ComputationPi`

Пока автор писал текст этого раздела, запущенное приложение `ComputationPi` выполнило пять миллиардов итераций и приблизилось к встроенному значению `Pi` в десятом разряде. Интересно, на сколько хватит терпения у вас?

Этот простой пример — первый шаг в освоении способов создания собственных классов, потомков базового класса `TThread`. Из-за своей простоты он не лишен недостатков; более того — если бы вычислительных нитей было несколько, кое-какие приемы оказались бы даже ошибочными. Но об этом — в следующем разделе.

13.4. Проблемы синхронизации потоков

К сожалению, простота создания потоков подчас "компенсируется" сложностью их применения. Две типичные проблемы, с которыми программист может столкнуться при работе с потоками, — это *тупики (deadlocks)* и *гонки (race conditions)*.

13.4.1. Тупики

Однажды автор наблюдал на трамвайной остановке следующую забавную картину (рис. 13.6).

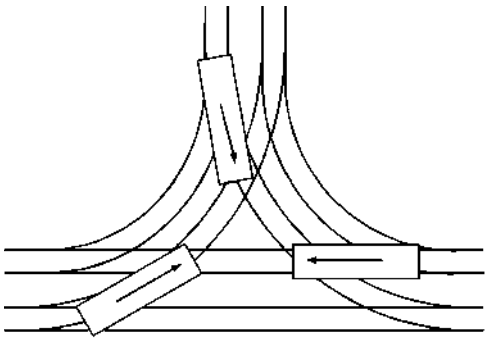


Рис. 13.6. Ситуации тупиков, возникающие не только в программировании

Рис. 13.6 дает исчерпывающее пояснение ситуации тупиков. Тупики возникают, когда поток ожидает ресурс, который в данный момент принадлежит другому потоку. Рассмотрим пример: Поток 1 захватывает ресурс А и, для того чтобы продолжать работу, ждет возможности захватить ресурс Б. В то же время Поток 2 захватывает ресурс Б и ждет возможности захватить ресурс А. Развитие действия по этому сценарию заблокирует оба потока; ни один из них не будет исполняться. Ресурсами могут выступать любые совместно используемые объекты системы — файлы, массивы в памяти, устройства ввода/вывода и т. п.

В ситуации, изображенной на рисунке, три трамвая захватили по одному ресурсу (перекрестку) и пытаются захватить еще один, что, очевидно, невозможно без освобождения уже захваченных. В жизни ситуация разрешилась просто — самый молодой из водителей был вынужден отъехать. В информационных технологиях все бывает сложнее. Откройте любой документ, сопровождающий очередной пакет обновления к любой версии Windows. Очень часто там можно найти информацию об одной-двух исправленных ситуациях тупиков.

13.4.2. Гонки

Ситуация гонок возникает, когда два или более потока пытаются получить доступ к общему ресурсу и изменить его состояние. Рассмотрим следующий пример. Пусть Поток 1 получил доступ к ресурсу и изменил его в своих интересах; затем активизировался Поток 2 и модифицировал этот же ресурс до завершения Потока 1. Поток 1 полагает, что ресурс остался в том же

состоянии, в каком был до переключения. В зависимости от того, когда именно был изменен ресурс, результаты могут варьироваться — иногда код будет выполняться нормально, иногда нет. Следовательно, программисты не должны строить никаких гипотез относительно порядка исполнения потоков, так как планировщик ОС может запускать и останавливать их в любое время.

```
Inc(i);
if i = iSomething then
 DoSomething;
```

Здесь `i` — глобальная переменная, доступная из обоих потоков. Пусть два или более потоков исполняют этот код одновременно. Поток 1 увеличил значение переменной `i` и хочет проверить ее значение для выполнения тех или иных действий. Но тут активируется другой поток, который еще увеличивает значение `i`. В результате первый поток "проскакивает" мимо условия (равенство `i` определенному значению), которое, казалось бы, должно было быть выполнено.

Возникновения как ситуаций гонок, так и тупиков можно избежать, если использовать приемы, обсуждаемые в следующем разделе.

13.5. Средства синхронизации потоков

Проще всего говорить о синхронизации, если создаваемый поток не взаимодействует с ресурсами других потоков и не обращается к VCL. Допустим, у вас на компьютере несколько процессоров, и вы хотите распараллелить вычисления. Тогда вполне уместен следующий код:

```
MyCompThread := TComputationThread.Create(False);
// делаем что-либо, пока второй поток вычисляет
DoSomeWork;
// теперь ожидаем его
MyCompThread.WaitFor;
```

Приведенная схема совершенно недопустима, если во время своей работы поток `MyCompThread` обращается к VCL посредством метода `Synchronize`. В этом случае поток ждет завершения главного потока для обращения к VCL, а тот, в свою очередь, его окончания — классический тупик!

За "спасением" следует обратиться к программному интерфейсу Win32. Он предоставляет богатый набор инструментов, которые могут понадобиться для организации совместной работы потоков.

Главные понятия, используемые при описании механизмов синхронизации, — *функции ожидания* и *объекты синхронизации*. В Windows API предусмотрен ряд функций, позволяющих приостановить выполнение вызвавшего

эту функцию потока вплоть до того момента, когда будет изменено состояние какого-то объекта, называемого объектом синхронизации. (Под этим термином здесь понимается не объект Delphi, а объект операционной системы.) Простейшая из этих функций — `WaitForSingleObject` — предназначена для ожидания одного объекта.

К возможным вариантам объектов синхронизации относятся объекты четырех типов, разработанные специально для синхронизации: *событие (event)*, *взаимное исключение (mutex)*, *семафор (semaphore)* и *таймер (timer)*.

Кроме специальных объектов, можно организовать ожидание и объектов других типов, дескриптор которых используется в основном для других целей, но может применяться и для ожидания. К ним относятся: процесс, поток, оповещение об изменении в файловой системе (change notification) и консольный ввод.

Косвенно к этой группе может быть добавлена *критическая секция (critical section)*.

Примечание

Перечисленные выше типы объектов синхронизации в основном инкапсулированы в состав классов Delphi. У программиста есть две возможности. С одной стороны, в состав библиотеки VCL включен модуль SYNCOBJS.PAS, содержащий классы для события (`TEvent`) и критической секции (`TCriticalSection`). С другой, с Delphi поставляется отличный пример IPCDEMOS, который иллюстрирует проблемы взаимодействия процессов и содержит модуль IPCTHRD.PAS с аналогичными классами — для того же события, взаимного исключения (`TMutex`), а также совместно используемой памяти (`TSharedMem`).

Перейдем к подробному описанию типов объектов, используемых для синхронизации.

13.5.1. Событие

Объект типа событие (event) — простейший выбор для решения задач синхронизации. Он подобен дверному звонку — звенит до тех пор, пока его кнопка находится в нажатом состоянии, извещая об этом факте окружающих. Такой объект может находиться в двух состояниях, а "слышать" его могут многие потоки сразу.

Класс `TEvent` (модуль SYNCOBJS.PAS) имеет два метода: `SetEvent` и `ResetEvent`, переводящих объект в активное и пассивное состояние, соответственно. Конструктор имеет следующий вид:

```
constructor Create(EventAttributes: PSecurityAttributes; ManualReset,
InitialState: Boolean; const Name: string);
```

где параметр `InitialState` — начальное состояние объекта, `ManualReset` — способ его сброса (перевода в пассивное состояние). Если параметр

ManualReset равен True, событие должно быть сброшено вручную. В противном случае событие сбрасывается тогда, когда стартует хотя бы один поток, ждавший данный объект.

На третьем методе:

```
TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError);
function WaitFor(Timeout: DWORD): TWaitResult;
```

остановимся подробнее. Он дает возможность ожидать активизации события в течение Timeout миллисекунд. Как вы могли догадаться, внутри этого метода происходит вызов функции WaitFotSingleObject. Типичных значений результата на выходе WaitFor два — wrSignaled, если произошла активизация события, и wrTimeout, если за время тайм-аута ничего не произошло.

Примечание

Если нужно (и допустимо!) ждать бесконечно долго, следует установить параметр TimeOut в значение INFINITE.

Рассмотрим маленький пример. Включим в состав нового проекта объект типа TThread, наполнив его метод Execute следующим содержимым:

```
Var res: TWaitResult;

procedure TSimpleThread.Execute;
begin
e := TEvent.Create(nil,True,false, 'test');
  repeat
  e.ReSetEvent;
  res := e.WaitFor(10000);
  Synchronize(ShowInfo);
  until Terminated;
e.Free;
end;

procedure TSimpleThread.ShowInfo;
begin
  ShowMessage(IntToStr(Integer(res)));
end;
```

На главной форме разместим две кнопки — нажатие одной из них запускает поток, нажатие второй активизирует событие.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 TSimpleThread.Create(False);
```

```
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
 e.SetEvent;
end;
```

Нажмем первую кнопку. Появляющийся на экране результат (вызов метода `ShowInfo`) будет зависеть от того, была ли нажата вторая кнопка или истекли отведенные 10 секунд.

События используются не только для работы с потоками — некоторые процедуры операционной системы автоматически переключают их. К числу таких процедур относятся *отложенный (overlapped) ввод/вывод* и события, связанные с коммуникационными портами.

13.5.2. Взаимные исключения

Объект типа взаимное исключение (Mutex) позволяет только одному потоку в данное время владеть им. Если продолжать аналогии, то этот объект можно сравнить с эстафетной палочкой.

Класс, инкапсулирующий взаимное исключение, — `TMutex` — находится в модуле IPCTHRD.PAS (пример IPCDEMOS).

Его конструктор

```
constructor Create(const Name: string);
```

задает имя создаваемого объекта. Первоначально объект не принадлежит никому. (Но функция API `CreateMutex`, вызываемая в нем, позволяет передать созданный объект тому потоку, в котором исключение произошло.)

Метод:

```
function Get(TimeOut: Integer): Boolean;
```

производит попытку в течение `TimeOut` миллисекунд завладеть объектом (в этом случае результат равен `True`).

Если объект больше не нужен, следует вызвать метод:

```
function Release: Boolean;
```

Программист может использовать взаимное исключение, чтобы избежать считывания из общей памяти и записи в нее информации несколькими потоками одновременно.

13.5.3. Семафор

Семафор (semaphore) подобен взаимному исключению. Разница между ними в том, что семафор может управлять количеством потоков, которые имеют к нему доступ. Для семафора устанавливается предельное число потоков,

которым доступ разрешен. Когда это число достигнуто, последующие потоки будут приостановлены, пока один или несколько потоков не отсоединятся от семафора и не освободят доступ.

Примером использования семафора может служить одновременная работа группы потоков с фрагментом совместно используемого пула памяти. Так как совместно используемая память допускает обращение к ней только определенного числа потоков, все прочие должны быть блокированы вплоть до того момента, когда один или несколько пользователей пула откажутся от его совместного использования.

13.5.4. Критическая секция

Работая в Delphi, программист может также применять объект типа критическая секция (critical section). Критические секции, по сути, подобны взаимным исключениям, однако между ними существуют два главных отличия.

- Взаимные исключения могут совместно использоваться потоками в разных процессах, а критические секции — нет.
- Если критическая секция принадлежит одному потоку, то другой, ожидающий поток, блокируется вплоть до освобождения критической секции. В отличие от этого, взаимное исключение разрешает продолжение выполнения ожидающего потока по истечении тайм-аута.

На первый взгляд, выигрыш от использования критической секции вместо взаимного исключения не очевиден. Критические секции, однако, более эффективны, чем взаимные исключения, так как используют меньше системных ресурсов. Взаимные исключения могут быть установлены на определенный интервал времени, по истечении которого выполнение продолжается; критическая секция всегда ждет столько, сколько потребуется.

Рассмотрим класс `TCriticalSection` (модуль SYNCOBJS.PAS). Логика использования его проста — "держать и не пущать". В многопоточном приложении создается и инициализируется общая для всех потоков критическая секция. Если один из потоков достигает критически важного участка кода, он пытается захватить секцию вызовом метода Enter:

```
MySection.Enter;
try
 DoSomethingCritical;
finally
 MySection.Leave;
end;
```

Когда другие потоки доходят до оператора захвата секции Enter и обнаруживают, что она уже захвачена, они приостанавливаются вплоть до освобождения секции первым потоком путем вызова метода `Leave`. Обратите внимание,

что вызов Leave помещен в конструкцию try..finally — здесь требуется стопроцентная надежность. Критические секции являются системными объектами и подлежат обязательному освобождению — впрочем, как и остальные рассматриваемые здесь объекты.

13.5.5. Процесс. Порождение дочернего процесса

Объект типа процесс (process) может быть использован для того, чтобы приостановить выполнение потока в том случае, если он для своего продолжения нуждается в завершении процесса. С практической точки зрения такая проблема возникает, когда нужно в рамках вашего приложения исполнить приложение, созданное кем-то другим, или, к примеру, сеанс MS-DOS.

Рассмотрим, как, собственно, один процесс может породить другой. Вместо устаревшей и поддерживаемой только для совместимости функции WinExec, перекочевавшей из прежних версий Windows, гораздо правильнее использовать более мощную:

```
function CreateProcess(lpApplicationName: PChar; lpCommandLine: PChar;
  lpProcessAttributes, lpThreadAttributes: PSecurityAttributes;
  bInheritHandles: BOOL; dwCreationFlags: DWORD; lpEnvironment: Pointer;
  lpCurrentDirectory: PChar; const lpStartupInfo: TStartupInfo;
  var lpProcessInformation: TProcessInformation): BOOL;
```

Первые два параметра ясны — это имя запускаемого приложения и передаваемые ему в командной строке параметры.

Параметр dwCreationFlags содержит флаги, определяющие способ создания нового процесса и его будущий приоритет. Использованные в приведенном далее примере (листинг 13.1) флаги означают: CREATE_NEW_CONSOLE — будет запущено новое консольное приложение с отдельным окном; NORMAL_PRIORITY_CLASS — нормальный приоритет.

Структура TStartupInfo содержит сведения о размере, цвете, положении окна создаваемого приложения. В листинге 13.1 используется поле wShowWindow: установлен флаг SW_SHOWNORMAL, означающий визуализацию окна с нормальным размером.

На выходе функции заполняется структура lpProcessInformation. В ней программисту возвращаются дескрипторы и идентификаторы созданного процесса и его первичного потока. Нам понадобится дескриптор процесса — в нашем примере создается консольное приложение, и затем происходит ожидание его завершения. "Просигналит" нам об этом именно объект lpProcessInformation.hProcess.

Листинг 13.1. Порождение дочернего процесса

```
var
      lpStartupInfo: TStartupInfo;
      lpProcessInformation: TProcessInformation;
begin
 FillChar(lpStartupInfo,Sizeof(lpStartupInfo),#0);
 lpStartupInfo.cb := Sizeof(lpStartupInfo);
 lpStartupInfo.dwFlags := STARTF_USESHOWWINDOW;
 lpStartupInfo.wShowWindow := SW_SHOWNORMAL;
  if not CreateProcess(nil,
                PChar('ping localhost'),
                nil,
                nil,
                false,
                CREATE_NEW_CONSOLE or NORMAL_PRIORITY_CLASS,
                nil,
                nil,
                lpStartupInfo,
                lpProcessInformation)
   then
    ShowMessage(SysErrorMessage(GetLastError))
   else
    begin
     WaitForSingleObject(lpProcessInformation.hProcess, 10000);
     CloseHandle(lpProcessInformation.hProcess);
    end;
end;
```

13.5.6. Поток

Поток может ожидать другой поток, точно так же, как и другой процесс. Ожидание можно организовать с помощью функций API (см. листинг 13.1), но удобнее это сделать при помощи метода `TThread.WaitFor`.

13.5.7. Консольный ввод

Консольный ввод (console input) годится для потоков, которые должны ожидать отклика на нажатие пользователем клавиши на клавиатуре. Этот тип ожидания может быть использован в *программе дуплексной связи (`Chat`).*

Один поток при этом будет ожидать получения символов; второй — отслеживать ввод пользователя и затем отсылать набранный текст ожидающему приложению.

13.5.8. Оповещение об изменении в файловой системе

Этот вид объекта ожидания очень интересен и незаслуженно мало известен. Мы рассмотрели практически все варианты того, как один поток может подать сигнал другому. А как получить сигнал от операционной системы? Ну, например, о том, что в файловой системе произошли какие-то изменения? Такой вид оповещения позаимствован из ОС Unix и доступен программистам, работающим с Win 32.

Для организации мониторинга файловой системы нужно использовать 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 реагированию на следующее изменение. Наконец, с помощью третьей функции следует закрыть ставший ненужным объект.

Так может выглядеть код метода `Execute` потока, созданного для мониторинга файловой системы:

```
var DirName : string;
...
procedure TSimpleThread.Execute;
var r: Cardinal;
  fn : THandle;
begin
 fn := FindFirstChangeNotification(pChar(DirName), True,
FILE_NOTIFY_CHANGE_FILE_NAME);
 repeat
 r := WaitForSingleObject(fn,2000);
 if r = WAIT_OBJECT_0 then
  Synchronize(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:

```
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. Для того чтобы оно стало полнофункциональным, предусмотрите в нем механизм перезапуска потока при изменении обследуемой папки.

13.6. Локальные данные потока

Интересная проблема возникает, если в приложении будет несколько одинаковых потоков. Как избежать совместного использования одних и тех же переменных несколькими потоками? Первое, что приходит на ум, — описать и использовать поля объекта — потомка `TThread`, которые можно добавить при его создании. Каждый поток соответствует отдельному экземпляру класса, и данные разных потоков пересекаться не будут. (Кстати, это одно из больших удобств использования класса `TThread`.) Но есть функции API, которые знать не знают о классах Delphi и их полях и свойствах. Для поддержки разделения данных между потоками на нижнем уровне в язык Object Pascal введена специальная директива — `threadvar`, которая отличается от директивы описания переменных var тем, что применяется только к локальным данным потока. Следующее описание:

```
Var
 data1: Integer;
```

```
threadvar
 data2: Integer;
```

означает, что переменная data1 будет использоваться всеми потоками данного приложения, а переменная data2 будет у каждого потока своя.

13.7. Как избежать одновременного запуска двух копий одного приложения

Такая задача возникает очень часто. Многие пользователи, особенно начинающие, не вполне понимают, что между щелчком по значку приложения и его запуском может пройти несколько секунд, а то и десятков секунд. Они начинают щелкать по значку, запуская все новые копии. Между тем, при работе с базами данных и во многих других случаях иметь более одной копии не только не нужно, но и вредно.

Идея заключается в том, чтобы первая создаваемая копия приложения захватывала некий ресурс, а все последующие при запуске пытались сделать то же самое и в случае неудачи завершались.

Пример такого ресурса — общий блок в файле, отображаемом в память. Поскольку этот ресурс имеет имя, можно сделать его уникальным именно для вашего приложения:

```
var UniqueMapping : THandle;
    FirstWindow : THandle;
begin
  UniqueMapping := CreateFileMapping($ffffffff,
  nil, PAGE_READONLY, 0, 32,'MyMap');
  if UniqueMapping = 0 then
  begin
    ShowMessage(SysErrorMessage(GetLastError));
    Halt;
   end
  else if GetLastError = ERROR_ALREADY_EXISTS  then
   begin
    FirstWindow := FindWindowEx(0, 0, TfmMain.ClassName, nil);
    if FirstWindow<>0 then
       SetForegroundWindow(FirstWindow);
    Halt;
   end;
// нет других копий — продолжаем
Application.Initialize;
...
```

Примерно такие строки нужно вставить в начало текста проекта, до создания форм. Блок совместно используемой памяти выделяется в системном страничном файле (об этом говорит параметр `MyMap` функции `CreateFileMapping`, равный −1). Код ошибки `ERROR_ALREADY_EXISTS`, полученный при создании блока, свидетельствует о наличии работающей копии приложения. В этом случае повторно запущенное приложение переключает фокус на главную форму другого экземпляра и завершается; в противном случае процесс инициализации продолжается.

Резюме

Потоки, как и другие мощные инструменты, должны использоваться с осторожностью и без злоупотреблений, так как могут вызывать ошибки, которые очень трудно найти. Есть очень много аргументов в пользу применения потоков, но есть и доводы против этого. Работа с потоками будет проще, если учитывать следующие положения.

- Если потоки работают только с переменными, объявленными внутри их собственного класса, то ситуации гонок и тупиков крайне маловероятны. Другими словами, избегайте использования в потоках глобальных переменных и переменных других объектов.
- Если вы обращаетесь к полям или методам объектов VCL, делайте это только посредством метода `Synchronize`.
- Соблюдайте чувство меры при синхронизации вашего приложения, иначе оно будет работать как один-единственный поток. Избыточно синхронизированное приложение теряет все преимущества от наличия нескольких потоков, так как они будут постоянно останавливаться и ждать синхронизации.

Потоки предоставляют изящное решение некоторых сегодняшних проблем программирования; но они также усложняют и без того непростой процесс отладки. И все же преимущества потоков однозначно перевешивают их недостатки.

Часть V

Данные XML в распределенных приложениях

Глава 14. Использование XML

Глава 15. Использование данных в формате XML

Глава 14

Использование XML

В Delphi 6 без малого три десятка страниц Палитры компонентов, и на каждой из них — по десятку, если не больше, компонентов. Редко о каком из них можно написать очень много — авторы книг обычно группируют их, чтобы получить приемлемое число глав.

Тем не менее, перед вами целая глава, полностью посвященная одному компоненту — TXMLDocument. Во-первых, термин "XML" слышали почти все, сам язык знают многие, но далеко не все представляют горизонты практического применения этого средства. Во-вторых, XML лежит в основе многих других технологий Delphi — без понимания азов подбираться к ним бесполезно.

Сначала мы рассмотрим основы языка, понятие XML-схем и объектную модель документа (DOM) и, наконец, — приемы работы с XML в среде Delphi.

14.1. Что такое XML и для чего он предназначен

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 (Standartized 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 гораздо более строг.

Но внешним сходством все и заканчивается. Главное отличие кроется в назначении 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 получении 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 ни было новых конструкций в HTML (напомним, текущая версия языка 4.0), 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 кодировки UNICODE. Это облегчает чтение различных документов и 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. Многие браузеры уже сейчас поддерживают XML.

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 конечно же, на XML. Эти языки и их поддержка в Delphi будут рассмотрены нами позднее.

В заключение вводной части приведем схему работы с XML-документом (рис. 14.1). Помимо самого документа, на входе системы должно присутст-

вовать его описание (определение). Надеемся, из прочитанного ясно, почему в отличие от HTML теги XML-документа нигде не стандартизованы и не описаны. По этой причине требуется дополнительная информация об 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 могут содержаться как в теле самого документа, так и в отдельном файле, на который документ должен ссылаться.

Рис. 14.1. Общая схема работы с XML-документами

Перед началом обработки документ 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 (DOM, Document Object Model)*. 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 используются 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, в Borland не стали "складывать все яйца в одну корзину". В Delphi входят также другие — альтернативные анализаторы, и предоставлена возможность подключения собственных. Аналогичные утилиты для других ОС и среды Java имеют Sun, Oracle, IBM и другие поставщики ПО.

Наконец, четвертое звено на рисунке — это, собственно, приложение, предназначенное для работы с XML; принципы разработки такого приложения описываются в этой главе. Но прежде рассмотрим первые три звена подробнее.

14.2. 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, 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 существует 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="2001" currency="USD">100000</salary>
```

Элемент salary (зарплата) имеет значение 100000, а атрибуты указывают, что выражена эта цифра в долларах и относится к 2001 году.

Элемент может не содержать ни значения, ни атрибутов. В этом случае допустима краткая форма его записи, в которой отсутствует закрывающий тег:

```
<dummy/>
```

Что есть 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 14.1).

Листинг 14.1. 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 дополнительную информацию о них.

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.

Важно подчеркнуть еще одно существенное отличие синтаксиса XML от HTML. В XML требуется, чтобы открывающему тегу *обязательно* соответствовал закрывающий. Отсутствие последнего является ошибкой — такой документ не будет обрабатываться. 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 использовать XML-анализаторы, старайтесь таких вольностей все же не допускать.

Перейдем теперь к общей структуре XML-документа, опираясь на содержимое листинга 14.1. XML-документ состоит из трех частей:

- пролога;
- определения (необязательно);
- собственно тела документа, начинающегося с корневого элемента.

14.2.1. 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 представляют собой инструкции анализатору для выбора тех или иных режимов или параметров. Как правило, XML-документ начинается с *объявления XML (XML declaration)*:

```
<?xml version="1.0"?>
```

Эта команда обработки, строго говоря, не обязательна, но рекомендована W3C, так как указывает 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, даже 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. Допустимым считается такой комментарий:

```
<!-- 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 искать в этом потоке зарезервированные символы, вроде <, >, и т. п.? Все содержимое, находящееся между `<![CDATA[` и `]]>` анализатор просто пропускает, не проверяя на соответствие синтаксису XML. Но это не означает, что туда можно писать вообще все, что угодно. Формат XML — текстовый, и символы с шестнадцатеричными значениями 0x0—0x1F относятся к запрещенным (за исключением символов табуляции (0x9), перевода строки (0xD) и возврата каретки (0xA)). Таким образом, если вы хотите поместить в XML двоичные данные, — скажем, результат сжатия данных или их шифрования — воспользуйтесь кодировкой Base64 или аналогичной *(подробнее об этом — в следующих главах)*.

14.2.2. Определение

Под определением имеется в виду DTD или XML-схема. Обе разновидности решают одну и ту же задачу — описание структуры документа; а именно, всех элементов с их атрибутами, типами данных, связями и уровнями вложенности.

Почему же 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, в правильном документе соблюдаются следующие три условия.

2. Он содержит, по крайней мере, один элемент.
3. В нем есть корневой элемент, уникальный для данного документа.
4. Все остальные элементы 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 с использованием интерфейсов объектной модели документа (DOM).

Состоятельный документ, во-первых, должен быть правильным, а во-вторых, он должен полностью соответствовать определению: своему DTD или схеме.

Анализатор фирмы Microsoft позволяет включать или выключать режим проверки на состоятельность: свойство `ValidateOnParse` указывает, выполнять ли такую проверку при загрузке документа. Одноименное свойство имеется в компоненте `TXMLDocument` — основе XML-стратегии Borland. Если вы планируете использование XML для систем автоматизированного документооборота, подчиненных жестким требованиям к формату данных, целесообразно сразу отбраковывать документы, не являющиеся состоятельными.

Изначально структура XML-документа описывалась при помощи определения DTD, позаимствованного у языка SGML. DTD представляет собой набор тегов, ограниченных символами <! и >. Вот так могло бы выглядеть DTD для нашего документа из листинга 14.1:

```
<!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 уступает свои позиции. Читателям, нуждающимся в деталях, рекомендуем документацию W3C.

Идея появления XML-схем проста и красива. Кто-то задался справедливым вопросом: а почему бы ни 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).

14.2.3. Тело документа. Корневой элемент

Наконец, о *корневом элементе (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 14.1 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, что помимо самого документа в XML-файле содержится также его схема. Схема обычно имеет собственный корневой элемент — `<Schema>`. В этом случае над двумя существующими элементами (корнем документа и корнем схемы) надстраивают еще один, который часто называют просто `<xml>` (не путайте с прологом `<?xml?>`).

Итак, резюмируем обзор 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 модель "дерева".

Нужно обязательно упомянуть здесь и о другом способе обработки XML. Параллельно с DOM ведется разработка еще одного стандарта, называемого SAX (Simple API for XML). В него заложена совсем другая модель. 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. Создавая обработчики этих событий, программист реализует с их помощью требуемую функциональность.

Такой подход кажется разумным, когда:

- анализируемый документ достаточно велик (мегабайты и более) и приходится останавливать весь процесс в ожидании, пока анализатор полностью обработает его;
- полный анализ документа не требуется; нужные решения можно принять, прочитав только начальную часть документа.

В состав анализатора фирмы Microsoft входят объекты, поддерживающие SAX, однако они не интегрированы Borland в Delphi. Интересующиеся читатели могут самостоятельно подключить их, используя существующие в Delphi возможности работы с объектами Автоматизации.

14.3. Объектная модель документа

Итак, анализатор проверил и загрузил правильный XML-документ. Теперь разработчик может обратиться к документу и получить в свое распоряжение набор интерфейсов, позволяющих работать с ним как с древовидной структурой, другими словами, объектную модель документа или DOM. Предоставление DOM — необходимая задача каждого анализатора; к счастью, усилия по стандартизации оказали на XML благотворное влияние.

Консорциум W3C постоянно работает над усовершенствованием DOM. Основой для работы всех анализаторов стал стандарт DOM level 1, принятый в 1998 г. В настоящий момент действует стандарт для 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. Для достижения переносимости модель DOM описана на языке описания интерфейсов IDL (Interface definition language), 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 фирмы Netscape / JScript фирмы 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, используя знакомые нам термины COM-объектов. Такое отображение сделано как Microsoft, так и Borland.

Соответственно, программист в Delphi для работы с DOM может применять целый ряд возможностей:

- использование "чистой" DOM (интерфейсы `IDOMNode/IDOMDocument` и производные от них);
- применение компонента `TXMLDocument` и интерфейсов `IXMLNode/IXMLDocument`, которые поддерживаются этим компонентом;
- обращение к интерфейсам и функциям анализатора напрямую; например, для анализатора MSXML они называются `IXMLDOMNode/IXMLDOMDocument`.

Это богатство возможностей не должно вас пугать. Все они базируются на стандартной модели DOM и немного отличаются синтаксисом и наличием дополнительных функций.

14.3.1. Интерфейсы семейства *IDOMNode*

Основная идея модели DOM — представить *весь* результат работы анализатора в виде вершин древовидной структуры, причем каждая вершина — это отдельный объект. 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 корневого элемента исходят вершины, соответствующие его дочерним элементам, и так далее. Атрибуты каждого элемента также представляются в виде отдельного набора его дочерних вершин. Но эти вершины — отдельный список вершин особого типа — не могут иметь дочерних.

DOM-представление нашего первого XML-документа из листинга 14.1, полученное при помощи программы, рассматриваемой далее, показано на рис. 14.2.

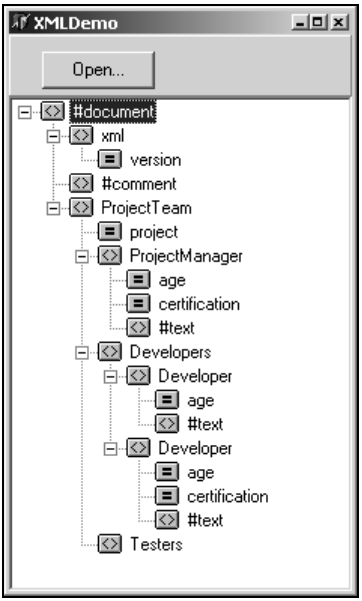


Рис. 14.2. Объектная модель нашего первого XML-документа, полученная при помощи программы, рассматриваемой далее

Здесь видна одна особенность DOM. У каждого элемента, содержащего текст, есть дочерняя вершина, представляющая этот текст. Эта особенность — типовая причина путаницы: программисты пытаются извлечь текст из свойств самого элемента, в то время как его надо искать в дочернем.

Примечание

Иначе устроен интерфейс `IXMLNode`, предложенный Borland, о котором будет рассказано далее.

Как сущности — любой вершине соответствует интерфейс `IDOMNode`. Она, а также ее специфические потомки, описаны в модуле xmldom. Рассмотрим их в обычном порядке — от общего к частному.

Свойства *nodeType*, *nodeName* и *nodeValue*

Для того чтобы не "заблудиться в трех соснах", рассмотрим свойства: `nodeType`, `nodeName`, `nodeValue`, которые есть у каждой вершины. Самое простое из них — `nodeType`, указывающее на тип вершины. В зависимости от его значения, интерпретация свойств `nodeName` и `nodeValue` может изменяться. Возможные варианты приведены в табл. 14.1.

Таблица 14.1. *Возможные значения свойств* `nodeType`, `nodeName` *и* `nodeValue` *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`

* Значок ↳ указывает на перенос слова на следующую строку в таблице.

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 следующих главах, будут рассмотрены подробнее.

Свойства и методы, управляющие другими вершинами

Для ориентации в структуре дерева, у интерфейса IDOMNode есть:

- 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;
- выделенные в отдельную структуру атрибуты (свойство attributes).

В табл. 14.2 дано краткое перечисление этих свойств и методов.

Таблица 14.2. *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 наличия дочерних вершин
`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

Таблица 14.2 *(окончание)*

Название	Назначение
`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 его равенстве `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;`	Проверяет, 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. 14.2 функции добавления/удаления дочерних вершин с атрибутами не работают, то у `IDOMNamedNodeMap` есть еще и аналогичные функции для атрибутов.

Просмотрев имеющиеся функции, вы наверняка заметили, что среди них не нашлось места функциям поиска. Неужели никто не додумался до этого раньше? Конечно, эта идея приходила в голову разработчикам, и не раз. Но спецификация поиска не входит в стандартную DOM и описывается отдельными документами. Мы обязательно еще будем об этом говорить.

Пространства имен

Имена элементов и атрибутов в XML записываются в текстовом виде. Это большое преимущество — документы понятны и удобочитаемы. Но недостатки, как говорится, — это продолжение достоинств. Предположим, что 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?

В XML у элементов и их атрибутов имя 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 в XML 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. 14.3).

Таблица 14.3. *Свойства, определяющие пространство имен*

Название	Описание
`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 `IDOMDocument` собраны методы, относящиеся к верхнему уровню его обработки.

Свойство

```
property documentElement: IDOMElement;
```

указывает на корневой элемент документа — в нашем примере это `'ProjectTeam'`. Свойство

```
property doctype: IDOMDocumentType;
```

предоставляет доступ к DTD документа, если таковое описание в нем содержится.

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 обособленные (standalone) вершины нужного типа, при создании не попадающие в дочерний список ни к одной из уже существующих вершин.

14.3.2. Пример создания приложения, использующего модель DOM

Теоретическая часть рассказа слегка затянулась, пора попробовать XML "на зуб". XML с его иерархиями вершин напоминает древовидную структуру — давайте же попытаемся загрузить документ в компонент типа `TTreeView`, чтобы увидеть эту структуру.

Поместите на форму компоненты `TXMLDocument` (со страницы **Internet** Палитры компонентов) и `TTreeView`, а также кнопку и диалог открытия файлов, свойство `Filter` у которого настройте на расширение xml (рис. 14.3). Добавьте список картинок `TImageList` с двумя картинками внутри — нам потребуется различать между собой элементы и атрибуты.

Далее введите в метод-обработчик события кнопки `OnClick` следующий код (листинг 14.2).

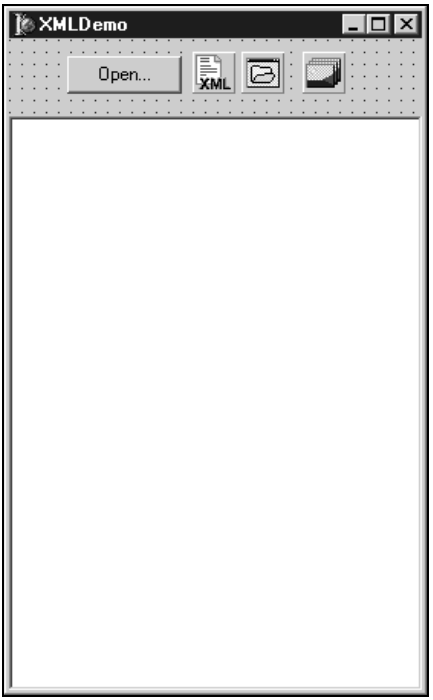


Рис. 14.3. Внешний вид главной формы простого XML-загрузчика

Листинг 14.2. Код приложения XMLDemo1

```
procedure TForm1.Button1Click(Sender: TObject);


function DOMNode2TreeViewNode(ADOMNode : IDOMNode; ParentNode :
TTreeNode):boolean;
var i: Integer;
    ThisNode : TTreeNode;
    TVNodes : TTreeNodes;
begin
 if not Assigned(ADOMNode) then Exit;
 TVNodes := tvXML.items;
 ThisNode := TVNodes.AddChild(ParentNode, ADOMNode.NodeName);
 if not Assigned(ThisNode) then Abort;
 ThisNode.ImageIndex := 0;
 if Assigned(ADOMNode.attributes) then
  for i := 0 to ADOMNode.attributes.length -1  do
```

```
   with TVNodes.AddChild(ThisNode, ADOMNode.Attributes.item[i].NodeName) do
    ImageIndex := 1;
 ThisNode.SelectedIndex := ThisNode.ImageIndex;
 if Assigned(ADOMNode.ChildNodes) then
  for i := 0 to ADOMNode.ChildNodes.length -1  do
   DOMNode2TreeViewNode(ADOMNode.ChildNodes[i], ThisNode);
end;


const
 SParseError = 'Parser error. Message: %s'#13#10'URL: %s; src :
%s;'#13#10'Line: %d, LinePos: %d';
begin
 if not OpenDialog1.Execute then Exit;
 XMLDocument1.LoadFromFile(OpenDialog1.FileName);
 try
  XMLDocument1.Active := True;
  tvXML.Items.Clear;
  try
   Screen.Cursor := crHourGlass;
   DOMNode2TreeViewNode(XMLDocument1.DOMDocument, nil);
  finally
   Screen.Cursor := crDefault;
  end;
 except
  on E:EDOMParseError do
    ShowMessage(Format(SParseError, [E.Reason, E.URL, E.srcText, E.Line,
E.LinePos ]));
end;
end;
```

В этом листинге основную работу делает рекурсивная процедура `DOMNode2TreeViewNode`:

- добавляет к дереву вершину, соответствующую текущей вершине (элементу) XML-документа;
- вставляет вершины, соответствующие атрибутам этого элемента, если они имеются;
- рекурсивно вызывает себя для всех дочерних вершин.

Пример выполняет простую работу — отображает файлы формата XML. Откройте в этой программе несколько файлов и посмотрите внимательно

на результаты ее работы — не заметили ли вы различий между содержимым файла и представлением дерева?

Да, одно такое отличие есть. Оно касается элементов, имеющих текстовое значение. Вы видите на рисунке множество вершин с именем #text? Эти вершины искусственно созданы анализатором — мы упоминали об этом при рассмотрении модели DOM.

14.4. Реализация модели DOM в Delphi

До появления Delphi 6 разработчикам, использующим XML, приходилось в основном работать с анализатором MSXML фирмы Microsoft. У него 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 фирмы Microsoft, он слишком тесно связан с другими частями ОС, в частности с Internet Explorer.

Примечание

Анализатор MS XML поставляется с Internet Explorer версий 4.01 и выше, а также со всеми новыми операционными системами, начиная с Windows 2000 и ME, и многими продуктами. Соответственно, на "голой" Windows 95 он устанавливаться и регистрироваться не будет — для этого нужно сначала установить IE соответствующей версии.

История изменений MSXML 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.

Совсем недавно вышел MSXML версии 4, вроде бы предназначенный для исправления ситуации. Из него чьей-то недрогнувшей рукой убраны все остатки совместимости с XSL. Он больше не называется просто анализатором — теперь это целый parsing suite (комплект средств для анализа). Соответственно продукт "потяжелел" до 5 Мбайт. К тому же он устанавливается только с помощью Windows Installer 2.0, исполняемые файлы (runtime) которого не входят даже в Windows 2000! Соответственно, прибавьте к размеру инсталляции вашего клиентского приложения еще 2 Мбайт и кучу трудностей. В общем, картина ясна: с точки зрения переносимости, MS XML — весьма спорный вариант.

Разработчики Delphi пошли по совершенно правильному пути, предусмотрев множественность реализаций DOM. В комплект поставки Delphi входят модули для использования трех анализаторов — MSXML, OpenXML и IBM XML. При этом MSXML является основным анализатором по умолчанию.

Если же вы хотите добиться максимальной совместимости (при минимальном использовании технологий фирмы Microsoft), то вам прямая дорога к анализатору OpenXML. Это свободно распространяемый в исходных кодах проект Дитера Кёлера (Dieter Köhler), чей сайт в Сети — **www.philo.de/xml**. В фирме Borland взяли за основу версию 2.3.14 (файл Sources\Internet\xdom.pas), слегка подправили и написали переходный модуль для подключения к общей инфраструктуре (oxmldom.pas).

К сожалению, вплоть до выхода пакета обновления Delphi 6.02 разработчики фирмы Borland не смогли обеспечить корректной совместимости третьего анализатора (фирмы IBM), и в настоящий момент он не может быть использован; но эта задача стоит у них на повестке дня.

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" — эта строка из русской сказки вспоминается, когда видишь иерархию 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`, `XMLDoc`. Пришло время разобраться кто есть кто и какую роль играет каждый из этих модулей.

Модуль *Xmldom*

Этот модуль 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`);
- механизмы поддержки списка реализаций DOM-анализаторов и регистрации подключаемых вами анализаторов.

Модуль *Msxml*

Этот модуль — трансляция на ObjectPascal библиотеки типов msxml.dll версии 2.0. Он содержит описания интерфейсов `IXMLDOMNode`, `IXMLDOMDocument` и т. п.

Если вы хотите использовать расширенные возможности более новых анализаторов фирмы Microsoft, создайте аналогичный модуль путем импорта библиотеки типов, содержащейся в msxml3.dll или msxml4.dll.

Модуль *Msxmldom*

Этот модуль не содержит новых интерфейсов (ура!), но в нем описаны классы (`TMSDOMNode` и производные), которые поддерживают семейство интерфейсов типа `IDOMNode`. Разумеется, это не "настоящие" классы, а переходники, осуществляющие вызов интерфейсов анализатора фирмы Microsoft. Кроме того, этот модуль берет на себя всю черновую работу по поиску анализатора MSXML подходящей версии.

Точно такую же роль связующего звена между Delphi и анализатором играют модули `Oxmldom` (OpenXML) и `Ibmxmldom` (IBM).

Модуль *XMLIntf*

`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*

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 которым столкнутся потенциальные пользователи 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 использовать анализатор напрямую или даже написать свой.

14.4.1. 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 использование

XML более удобным. По сравнению с DOM в интерфейсе вершины `IXMLNode` есть 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`;
- кроме этого, к обоим спискам добавлены методы доступа (accessors), знакомые Delphi-программистам по классу `TStrings`. Свойство

  ```
  property ChildValues[const IndexOrName: OleVariant]: OleVariant;
  ```

  является свойством вершины по умолчанию. Это псевдо-массив, 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;
  ```

  обеспечивает доступ к атрибутам элемента по их именам.

О свойстве `Text` следует поговорить отдельно, чтобы избежать возможных заблуждений.

Значение этого свойства определено не для всех вершин — есть 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 для чтения.

Итак, правило: свойство `Text` определено для элемента только в том случае, если у него есть ровно одна дочерняя вершина, 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? Ниже приведена часть примера, который еще будет рассмотрен 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 будущую текстовую 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. Однако анализаторы всегда ставят текст на первое место, и этот вариант возможен только в результате программистских манипуляций, поэтому мы его не учитываем.

```
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 со значением свойства `NodeValue`.

14.4.2. Взаимосвязь между всеми интерфейсами

После обзора нескольких уровней интерфейсов, следует рассказать о взаимосвязи между ними. Зачем нужен этот путь от общего к частному? Унифицированный доступ фирмы Borland хорош, но иногда нужно воспользоваться конкретными специфическими возможностями анализатора.

Особенно это касается анализатора фирмы Microsoft. При всех недостатках (размер и переносимость), функционально он — самый полный. Разработчики из Редмонда постоянно опережают стандарты W3C и добавляют к своей реализации интерфейсов DOM новые, подчас нестандартные возможности. Не будем участвовать в споре о том, хорошо это или плохо, — важно, что доступ к ним однозначно необходим.

Итак, какова же 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 одноименный класс Delphi, поддерживающий (implementing) этот интерфейс, также как и несколько других: `IDOMNodeEx`, `IDOMNodeSelect`, а также `IXMLDOMNodeRef` (в случае анализатора фирмы Microsoft) или `IOXDOMNodeRef` (для OpenXML). "Ref" в конце названия следует понимать буквально: единственное назначение последних двух интерфейсов — предоставлять ссылку на вершину более низкого уровня, непосредственно поддерживаемую анализатором.

  Классы Delphi, поддерживающие два последних интерфейса — `TMSDOMNode` или `TOXDOMNode` соответственно. Обращаясь к `IDOMNode`, вы на самом деле обращаетесь к экземпляру такого класса, а следовательно, можете применить правила присвоения типов (typecast):

```
MSDOMNode := (DOMNode as IXMLDOMNodeRef).GetXMLDOMNode;
```

  или

```
OXDOMNode := (DOMNode as IOXDOMNodeRef).GetDOMNode;
```

Взаимосвязи интерфейсов разных уровней для анализатора фирмы Microsoft показаны на рис. 14.4.

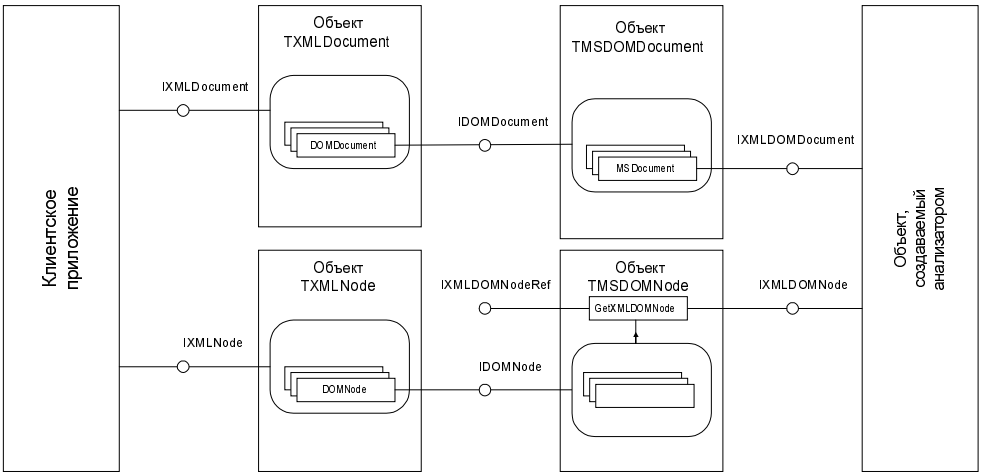


Рис. 14.4. Взаимосвязи интерфейсов и объектов, поддерживающих XML, в Delphi

Нужно иметь в виду, что обратное движение от дочерних вершин к родительским невозможно: имея вершину типа `IDOMNode`, отыскать соответствующую ей `IXMLNode` очевидными методами нельзя. Поэтому заранее планируйте свой проект так, чтобы работать с интерфейсами только одного уровня.

14.4.3. Загрузка XML

Разработчики ставили перед собой задачу сделать компонент `TXMLDocument` настолько универсальным, насколько возможно. Это в полной мере относится к способам загрузки XML. Возможные способы загрузки таковы.

- Из файла
  - 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 изменяется `FileName` и `Active`).
- Из потока — с помощью метода `LoadFromStream`.
- Из строки, содержащей текст на XML, — с помощью `LoadFromXML`.
- Путем присвоения значения свойству `XML`. Для активизации документа в этом случае нужно явно установить `Active = True`.

Очевидно, 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.

Напротив, установка `Active` в `False` уничтожает все внутренние объекты для поддержания DOM и присваивает указанным свойствам значение `nil`.

В одних случаях `Active` устанавливается автоматически, в других 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 связующие 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, так как его содержимое перестает соответствовать новому значению `XML`.
3. Изменение свойства XML всегда устанавливает свойство `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`.

Примечание

В версии Delphi 6.02 метод `Refresh` в ряде случаев завершается аварийно. Чтобы избежать этого, вместо вызова `Refresh` просто установите `Active` в `False` и затем снова в `True`.

6. При загрузке с помощью методов `LoadFromXML` и `LoadFromStream` значение свойства `XML` соответствует загруженной строке или потоку; при этом значение `FileName` не изменяется.
7. Если вы хотите автоматически сохранять сделанные вами изменения, установите параметр `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 упомянутый параметр установлен, 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 (см. п. 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. п. 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 фирмы Microsoft есть асинхронный режим работы, и поддержка его включена в `TXMLDocument`.

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`).

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

Для создания экземпляра компонента `IXMLDocument` в режиме выполнения помимо перечисленных способов у вас в распоряжении есть 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;
```

Вам предоставляется интерфейс `IXMLDocument`, но фактически создается экземпляр класса `TXMLDocument`. После окончания использования его можно

уничтожить, просто присвоив nil полученному от функции интерфейсу. Для экземпляров TXMLDocument, которые вы помещали на форму или в модуль данных, присваивать nil интерфейсам 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.

14.4.4. Обработка ошибок анализатора

Обработка ошибок также не была пущена на самотек специалистами, разрабатывавшими DOM. Среди интерфейсов объектной модели — интерфейс IDOMParseError, предоставляющий достаточно подробную информацию о возникшей при анализе документа ошибке.

Пара свойств:

```
property errorCode: Integer;
property reason: DOMString;
```

отвечает на вопрос "что произошло?". Разумеется, численные значения кодов ошибок (свойство errorCode) и их тексты у каждого анализатора свои. Коды ошибок анализатора фирмы 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;
```

содержат ответ на 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. Этот флаг указывает, 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 новый вопрос — где взять интерфейс IDOMParseError 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 EDOMParseError. Пример их использования — рассмотренный нами в листинге 14.2 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.

Как все-таки получить доступ к самому интерфейсу IDOMParseError? Если вы работаете с компонентом 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.

Исключительная ситуация DOMException возникает, когда та или иная возможность не поддерживается текущим анализатором (свойство DOMVendor). Например, такая исключительная ситуация инициируется 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 для чтения или создании вершины недопустимого типа. Числового кода ошибки и дополнительных свойств этот класс, описывающий исключительную ситуацию, 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 фирмой Borland, приведем еще один пример. Давайте посмотрим, как с помощью описанных интерфейсов создать простой редактор XML. Кстати, такой пример может представлять не только познавательный интерес. Свободно распространяемых редакторов XML не так уж много; проект XML Notepad, разрабатывавшийся в фирме Microsoft, закрыт.

К тому же, процедуры, управляющие вершинами дерева XML (добавление, удаление, перемещение, переименование и т. п.), несколько запутаны и трудны для понимания даже опытного программиста. Ничего лучшего, чем редактор, для знакомства с ними придумать нельзя (листинг 14.3).

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 используется для редактирования свойства `Text` и списка атрибутов текущей вершины, другая отображает собственно XML (содержимое одноименного свойства `IXMLNode`).

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 фирмы Microsoft.

Внешний вид окна нашего редактора показан на рис. 14.5.

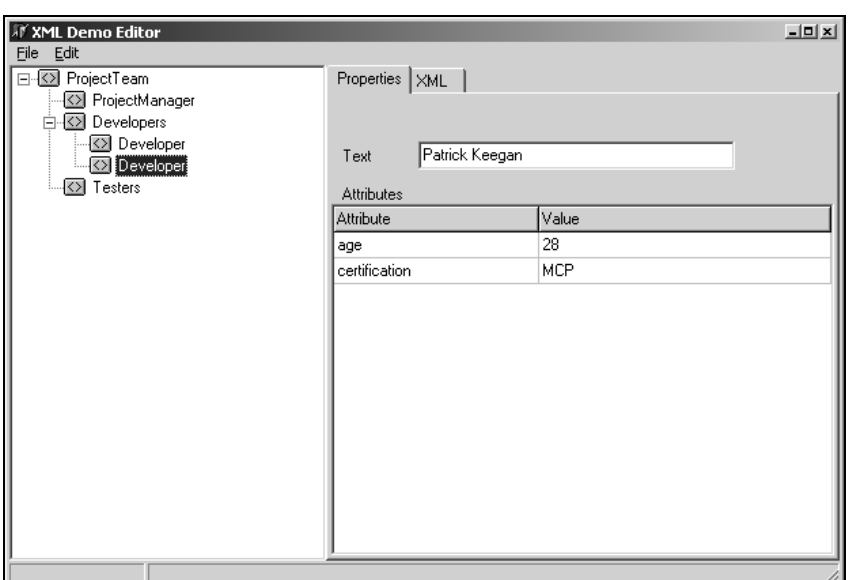


Рис. 14.5. Внешний вид окна демонстрационного редактора XML

Полный листинг примера не более 500 строк — размер, превышение которого, на наш взгляд, превращает пример в реальное приложение. Поэтому в нем реализованы только основные функции — чтение/запись, редактирование, работа с Буфером обмена. Вы можете добавить функции, необходимые лично вам.

Листинг 14.3. 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 XML-вершины нет ни одного, позволяющего переименовывать вершины. Это неудивительно, так как фундаментальное свойство `NodeName` доступно только для чтения. Для "переименования" нужно:

- создать новую вершину с желаемым именем;
- скопировать в нее все атрибуты и дочерние вершины;
- заменить старую вершину при помощи функции `ReplaceNode`.

Эту последовательность действий вы найдете в методе `TfmMain.tvXMLEdited`.

14.5. Анализатор MSXML, или Microsoft XML Core Services

К моменту написания этой книги текущей версией анализатора была версия 4. Ее основные отличия от предыдущих версий таковы.

- Появилась поддержка языка схем XSD (XML Schema Definition, **www.w3.org/TR/xmlschema-1**).
- Появилась поддержка объектной модели схемы (SOM, Schema Object Model) — нового набора интерфейсов, схожего с DOM, но предназначенного для работы с вершинами схемы.
- Исчезла поддержка языка XSL (которая сохранялась в версиях 2 и 3), теперь XSL полностью вытеснен усовершенствованной версией XSLT.
- Удалены независящие 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 MSXML. 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, MSXML версии 2 — первый из упоминающихся; очевидно, при распространении приложений с использованием MSXML вы тоже должны обеспечить наличие этой версии. Динамическая библиотека анализатора версии 2, а также последовавшей за ней 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 специалисты Microsoft решили, что самое время переименовать саму динамическую библиотеку в msxml2.dll. Эта версия библиотеки никогда не была доступна для отдельной загрузки из Internet и включения в состав дистрибутивов и оказалась самой короткоживущей.

Примечание

Мы упоминаем здесь названия только одной dll, однако это не означает, что анализатор состоит только из нее. В его состав входят еще одна или две библиотеки ресурсов, подлежащих интернационализации (процесс настройки, обеспечивающий работу на различных национальных языках и с учетом разных национальных стандартов). 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 фирмы Microsoft — Internet Explorer, который, напомним, поставлялся с MSXML версии 2. Для этого 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 MSXML, 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 загрузки с сайта Microsoft.

Однако такое вольное обращение с продуктом, выпущенным многомиллионным тиражом, не могло не вызвать шквала проблем. Многие WEB-приложения отказывались работать, пользователи и специалисты путались в режимах работы. К выходу следующей версии было принято решение об отступлении: версия 4 (динамическая библиотека msxml4.dll), как уже упоминалось, более не поддерживает независящих от версии идентификаторов.

Примечание

Интересно, что бета-версия MSXML 4 еще поддерживала их. Если вы устанавливали ее на свой компьютер, обязательно удалите ее!

Предположим, у вас на компьютере есть MSXML версий 2, 3 и 4 (вполне нормальная ситуация для разработчика), рассмотрим подробно ситуацию с идентификаторами с точки зрения программиста на Delphi.

Напомним, что идентификатор класса (CLSID) — это уникальный идентификатор, указывающий на класс COM-объекта, а программный идентификатор (ProgID) — его текстовый синоним, придуманный первоначально для пользователей Visual Basic.

В Delphi можно создать объект, вызвав функцию `CreateCOMObject`:

```
Var Doc : IXMLDOMDocument2;
...
 Doc := CreateCOMObject(CLASS_DOMDocument) as IXMLDOMDocument2;
```

В этом случае вы получите ссылку на нужный вам интерфейс, поддерживаемый созданным объектом 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 фирмы Microsoft поддерживала объект DOM-документа с идентификатором класса `CLASS_DOMDocument` и программными идентификаторами `'MSXML.DOMDocument'`, `'Microsoft.XMLDOM'` и `'Microsoft.XMLDOM.1.0'` (из трех чаще всего использовался первый).

С появлением в версиях 2.6 и 3 интерфейса `IXMLDOMDocument2` появились `CLASS_DOMDocument26`, `CLASS_DOMDocument30` и программные идентификаторы: `'MSXML2.DOMDocument'` (не зависящий 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 не зависящие 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 или более старой.

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 фирмы Microsoft. Если вы собрались положиться на его возможности, вам придется распространять с вашим продуктом дистрибутив анализатора (около 5 Мб) и, скорее всего, инсталлятор Windows Installer версии 2 (он также еще не включен в состав ОС Microsoft вплоть до Windows 2000).

Как создать экземпляр класса, поддерживаемого анализатором, и как называется этот объект?

Для получения описаний нужных интерфейсов вы можете подключить уже готовый файл MSXML.PAS из папки Source\RTL\Win\ в Delphi, который соответствует версии 2.0. Можно пойти 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 на страницу **ActiveX** Палитры Компонентов около 40 COM-объектов, 90% из которых вам вряд ли когда-нибудь понадобятся.

В качестве примера рассмотрим взаимодействие двух технологий фирмы Microsoft: XML и ADO *(ActiveX Data Objects, рассматривалась нами в гл. 7)*. 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. Как известно, компоненты и классы Delphi из соображений совместимости построены на базе библиотек ADO версии 2.1. Тем не менее, вполне вероятно, что на вашем компьютере имеется более новая версия. Версия 2.5 поставляется вместе с Delphi 6 Enterprise (файл MDAC_TYP.EXE). Множество приложений фирмы Microsoft устанавливают и используют ADO: Office, Internet Explorer, Visual Studio и т. п. Наконец, всегда можно загрузить последнюю версию ADO с сайта Microsoft, расположенного по адресу **www.microsoft.com/data**.

Если вы точно не знаете, какие версии ADO устанавливались на вашем компьютере, проще всего открыть редактор реестра regedit.exe и среди

дочерних ключей раздела HKEY_CLASSES_ROOT найти все, начинающиеся на 'ADODB.Recordset'. Если среди них есть 'ADODB.Recordset.2.5' или выше (текущая последняя версия 2.7), 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, который поставляется с Delphi в качестве примера использования баз данных от Microsoft Access (он устанавливается по умолчанию в папку C:\Program Files\Common Files\Borland Shared\Data\). Еще два компонента — TDataSource и TDBGrid — понадобятся нам для отображения содержимого этой базы в "традиционном" виде.

Раскрывающийся список будет содержать перечень всех таблиц в подключенной базе (это хорошо знакомые всем примеры Country, Employees, Parts и т. п.).

Наконец, компонент TMemo будет использоваться для хранения текста XML, а две кнопки — собственно для преобразования (рис. 14.6). Приведем в листинге 14.4 только код преобразования — полный код примера есть на прилагаемой к книге дискете.

Листинг 14.4. 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;
```

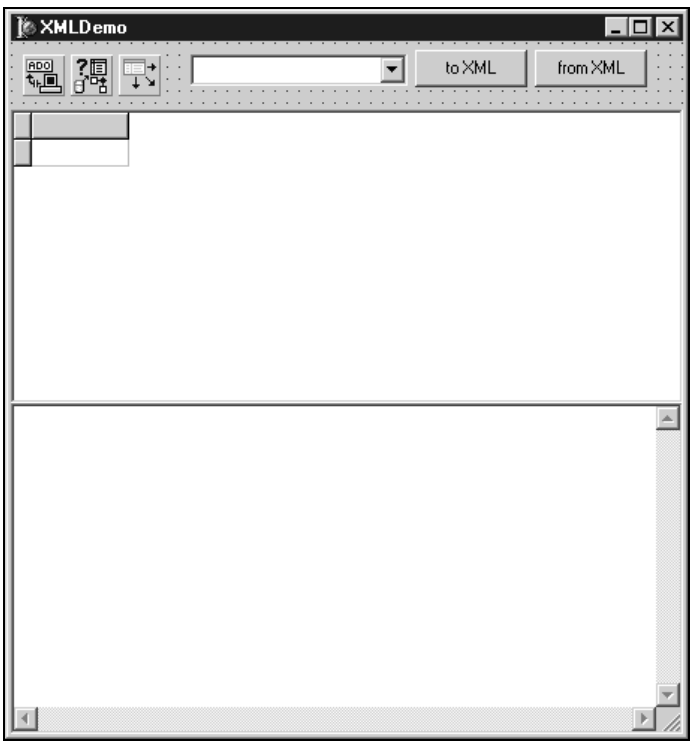


Рис. 14.6. Внешний вид главной формы примера XMLDemo3

Нажав кнопку **To XML**, вы превратите набор данных в документ, который можно редактировать. 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 преобразование. 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 фирмы Microsoft

широко использует этот интерфейс: 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` (напомним, версии 2.1), такой полиморфизм метода `Save` еще неизвестен. 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` (листинг 14.4).

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` (импортированную библиотеку типов ADO 2.1) 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 на то, что в примере не используется ни один из модулей работы с XML из состава Delphi (`Xmldom`, `Msxml`, `Msxmldom`, `Xmlintf`, `Xmldoc`).

Резюме

Мы только начали разговор об XML, языке, который перемещается все ближе к центру стратегии ведущих игроков мира IT. Задачей этой главы было познакомить вас с:

- синтаксисом и основами применения XML;
- объектной моделью DOM как основой программного доступа к элементам XML-документа;
- видами XML-анализаторов и компонентами Delphi — надстройками над анализаторами.

Этот разговор принципиально важен для Delphi; он продолжается во всех последующих главах. Сразу за этой главой XML рассматривается как средство для представления наборов данных. Главы 17 и 18 расскажут вам о подмножестве XML — языке WSDL и Web-службах. Глава 19 посвящена WebSnap — технологии фирмы Borland, также тесно связанной с XML.

Глава 15

Использование данных в формате XML

Delphi предоставляет разработчикам возможность использовать в приложениях данные в формате XML. Это могут быть любые типы приложений, в том числе приложения баз данных, распределенные приложения, Web-приложения.

Набор инструментов и компонентов обеспечивает 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 вы можете использовать специальную утилиту — ***XML Mapper***. Во время выполнения всю необходимую работу выполнят компоненты преобразования данных со страницы **Data Access** Палитры компонентов.

В этой главе обсуждаются следующие вопросы:

- что необходимо для использования данных XML;
- каким требованиям должны удовлетворять схемы XML;
- взаимодействие форматов XML и пакетов данных Delphi;
- назначение утилиты XML Mapper;
- применение компонента преобразования данных XML — `TXMLTransform`;
- использование компонентов доступа к данным XML.

15.1. Преобразование данных в формате XML

В Delphi реализован механизм преобразования данных в формате XML во внутренний формат пакетов данных. Напомним, что пакеты данных широко используются при передаче данных между клиентом и сервером в распределенных

приложениях. *(Более подробную информацию об этом см. в ч. 3 "Распределенные приложения баз данных".)*

Почему выбран именно такой способ применения данных XML?

Ответ прост. Использование пакетов данных позволяет набору компонентов трансформации XML легко взаимодействовать с практически любыми типами приложений Delphi. Благодаря тому, что данные XML преобразуются для применения в компонентах клиентских наборов данных, 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 применить уже хорошо знакомые.

В конце этой главы мы рассмотрим пример, в котором буквально несколькими действиями и при помощи немногих компонентов (большинство из которых вам хорошо знакомо по другим главам 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 преобразования и использования данных XML.

15.1.1. Схема преобразования данных XML

Итак, попробуем решить следующую задачу. В качестве исходных данных рассматривается файл XML, содержащий некоторые данные. *(Более подробную информацию о формате XML см. в гл. 14.)* 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. 15.1).

На основе исходного файла XML при помощи средств Delphi создается специальный ***трансформационный файл*** (или файл трансформации), в котором описываются связи между узлами XML и полями пакета данных (попробуйте угадать, на каком языке создается это файл — конечно на XML), а также структура результирующего документа.

Если преобразование будет сохраняться в новом пакете данных, то связывание можно выполнить один к одному, то есть для каждого узла XML можно создать поле пакета данных. Если используется существующий пакет данных, необходимо связать узлы XML только с существующими полями пакета.

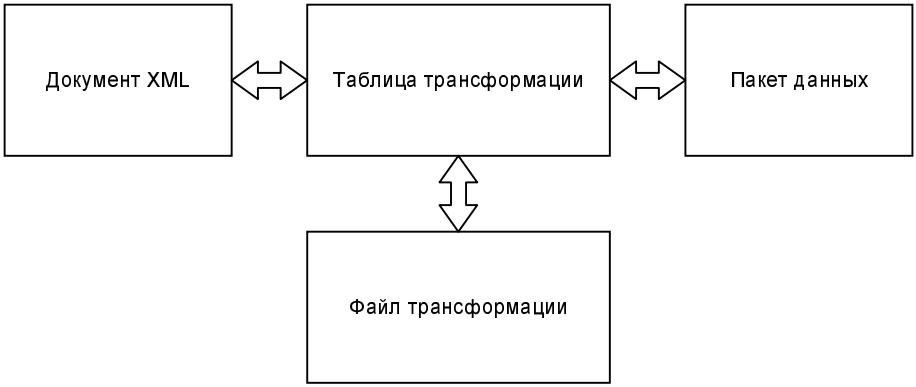


Рис. 15.1. Схема преобразования данных XML в формат пакета данных Delphi

После преобразования пакет данных можно использовать в приложении или сохранить в файле. Для этого применяется метод

```
procedure SaveToFile(const FileName: string = '';
Format TDataPacketFormat=dfBinary);
```

компонента клиентского набора данных. Формат файла зависит от значения параметра `Format`. При значении `dfXML` данные будут сохранены в формате XML.

Следует отметить, что описанная 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.

15.1.2. Формат пакета данных Delphi

Пакеты данных Delphi используются при обмене данными между клиентским набором данных и компонентом-провайдером *(более подробную информацию об этом см. в гл. 9)*. Для доступа к пакету данных обычно применяется свойство типа `OleVariant`. Пакет данных может быть сохранен в файле XML при помощи метода

```
procedure SaveToFile(const FileName: string = ''; Format TDataPacketFor-
mat=dfBinary);
```

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, пакет данных для таблицы `Country` демонстрационной базы данных Delphi выглядит так (листинг 15.1).

Листинг 15.1. 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">
```

Она представляет собой описание полей, включенных в состав набора данных. Для этого используется элемент `<FIELDS>`. Если в пакете имеются реальные данные, то они описываются в элементе `<ROWDATA>` в виде записей. Для каждой записи создается отдельный элемент `<ROW>`.

15.1.3. Инструментарий преобразования данных XML

Для создания трансформационных файлов и переноса данных на этапе разработки в состав Delphi включена *утилита XML Mapper*.

Использование функций преобразования файлов XML в приложениях обеспечивается набором компонентов со страницы **Data Access** Палитры компонентов.

Непосредственно для трансформации данных предназначен компонент `TXMLTransform`. Он может использоваться как для перевода 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. Этот компонент выполняет формирование и передачу пакетов данных клиентскому приложению и прием от него сделанных изменений

Компонент `TXMLTransformClient` является контейнером для компонента `TXMLTransform` и обеспечивает преобразование данных, полученных от компонента-провайдера `TXMLTransformProvider`.

Кроме этого на странице **Internet Express** Палитры компонентов имеется компонент `TXMLBroker`, который получает пакеты данных от компонента-провайдера (в том числе и от удаленного сервера приложения) и преобразует их в формат XML для *компонентов-продюсеров (более подробную информацию о компонентах-продюсерах см. в гл. 19).*

А теперь подробнее познакомимся с перечисленными инструментами и посмотрим, как они используются при создании приложений.

15.2. Утилита XML Mapper

Утилита XML Mapper вызывается из пункта меню **Tools | XML Mapper**. Она обеспечивает создание трансформационных файлов, схем XML, файлов с данными (как XML, так и пакетов данных). В дальнейшем эти файлы можно использовать в приложениях. Например, трансформационные файлы применяются в работе компонентов, выполняющих преобразование данных.

В соответствии со схемой преобразования данных XML (см. рис. 15.1) окно XML Mapper разделено на три панели (рис. 15.2).

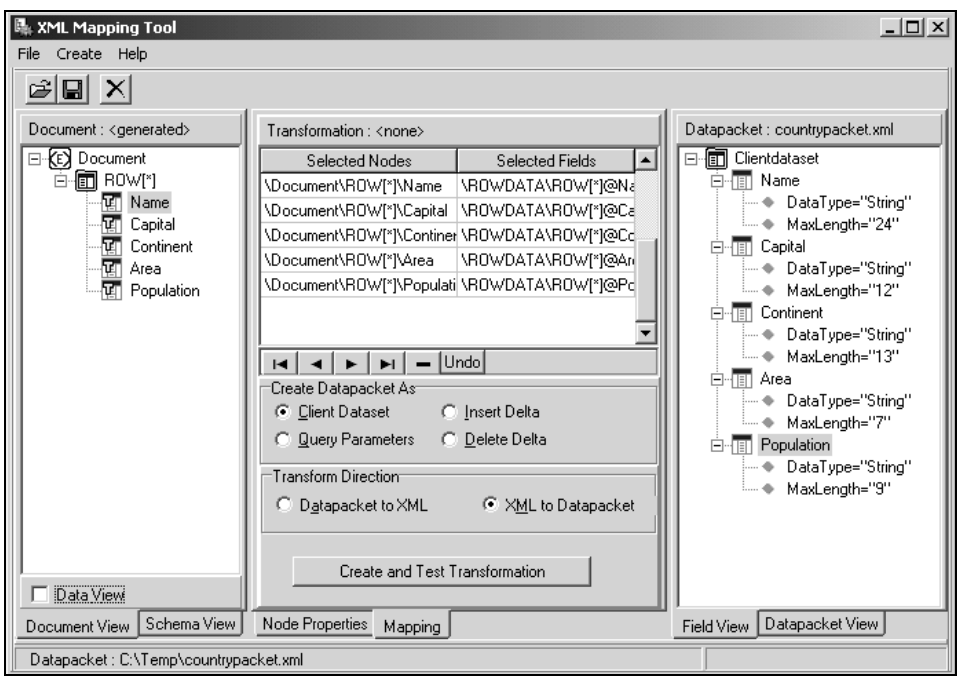


Рис. 15.2. Утилита XML Mapper, применяемая для создания трансформационных файлов

Левая панель предназначена для работы с документом XML, который является исходным в цепочке преобразований. Выбор файла и его загрузка осуществляется стандартными элементами управления. Здесь на вкладке **Document View** можно работать с деревом узлов и элементов документа XML. На этой панели выбираются элементы, предназначенные для преобразования. Вкладка **Schema View** используется 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.

15.2.1. Выбор исходного файла

Утилита XML Mapper работает со следующими типами данных:

- формат XML;
- формат пакета данных Delphi.

Информация о настроенных связях трансформации и вспомогательные данные содержатся в файлах следующих типов (данные в них хранятся в формате XML):

- трансформационный файл (XTR), включающий в себя карту связей преобразования и описание обоих наборов данных — XML и пакета данных;
- файл репозитория (XRP);
- файл схемы XML (DTD, XDR, XSD), позволяющий просмотреть структуру файла XML.

XML Mapper позволяет осуществлять трансформацию данных в обоих направлениях — из XML в пакет данных и обратно, поэтому можно открывать исходный файл как на левой, так и на правой панелях этой утилиты. Для этого используется одно из стандартных средств Windows — главное меню, кнопка панели инструментов, всплывающее меню панели.

Кроме этого, пакет данных можно получить напрямую от удаленного сервера. Для этого используется команда **Connect To Remote Server** всплывающего меню левой панели. После ее вызова появляется панель (рис. 15.3), на которой необходимо выбрать из списка нужный сервер (он должен быть зарегистрирован на компьютере) и набор данных этого сервера.

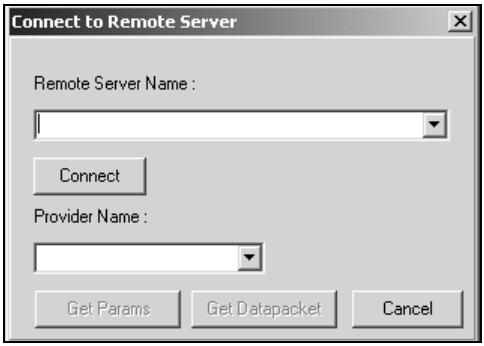


Рис. 15.3. Панель загрузки пакета данных с удаленного сервера

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).

15.2.2. 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 выбрать следующий пункт меню **Create**:

- **Datapacket from XML** — для создания пакета 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 **XML from Datapacket**. В нем вы можете изменить имена корневого элемента и других элементов (рис. 15.4).

Если на основе документа XML создается пакет данных, можно выбрать его тип. Для этого используется группа радиокнопок **Create Datapacket As** на вкладке **Mapping** центральной панели утилиты (см. рис. 15.2).

После этого созданную структуру можно сохранить в файле с помощью обычных средств.

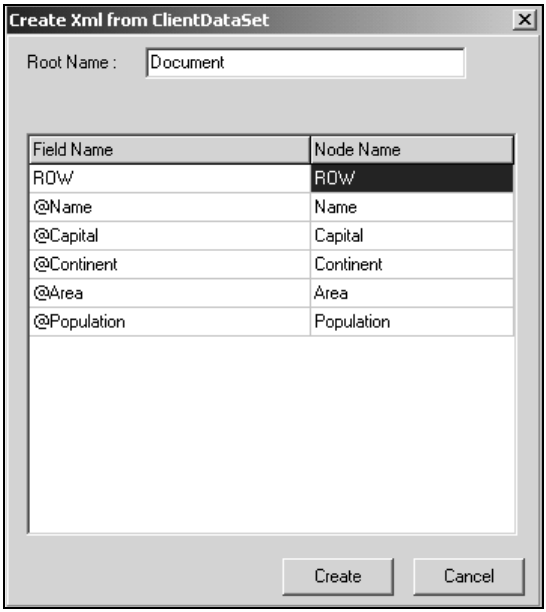


Рис. 15.4. Окно настройки элементов создаваемого документа XML

Примечание

Обратите внимание на то, что описываемые здесь действия формируют только структуру данных, без переноса реальных данных. Собственно преобразование данных рассматривается ниже.

15.2.3. 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 на левой или правой панели и дважды щелкнуть на нем мышью. В результате, на 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 заполнена 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

Учтите, что строки таблицы заполняются сверху вниз независимо от порядка следования элементов или полей на боковых панелях.

После связывания всех требуемых полей и элементов XML таблица связывания будет содержать модель преобразования данных.

15.2.4. Создание трансформационного файла и преобразование данных

Когда таблица связей будет заполнена, можно приступать к созданию трансформационного файла. Для этого необходимо выбрать команду **Transformation** из пункта меню **Create**. После этого в пункте меню **File | Save** становится доступна команда **Transformation**. Выбрав ее, можете сохранить созданный трансформационный файл и затем использовать его в работе с компонентами трансформации.

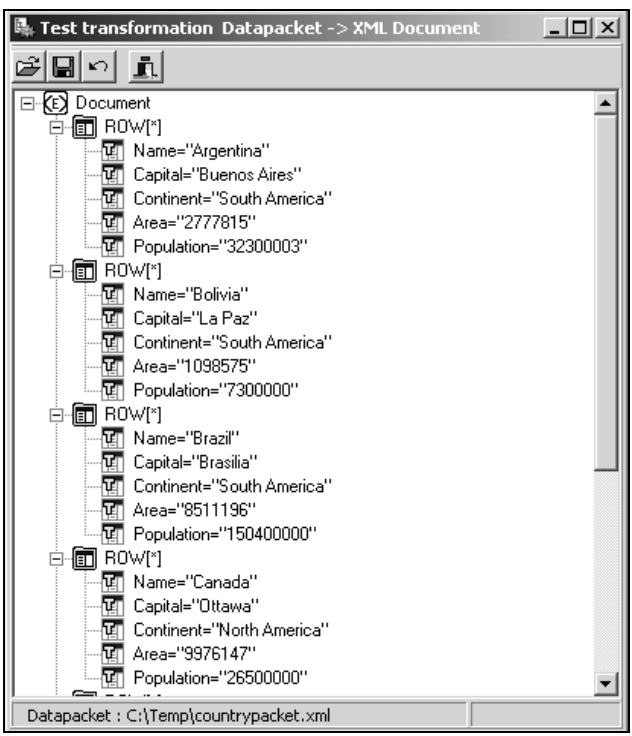


Рис. 15.5. Окно результата преобразования данных для документа XML

Помимо создания файла трансформации 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 (см. ранее) необходимо нажать кнопку **Create and Test Transformation** на вкладке **Mapping** центральной панели. Направление преобразования данных задается в группе радиокнопок **Transform Direction**.

Затем выполняется собственно преобразование данных, и его результат появляется в новом окне. Если приемником данных выступает документ XML, результат представляется в виде дерева XML (рис. 15.5). Для пакета данных результат отображается в виде таблицы.

15.3. Преобразование данных XML в распределенных приложениях

Теперь обратимся к программным аспектам использования данных 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, так как используется и в других компонентах, о которых речь пойдет далее.

В соответствии со схемой трансформации (см. рис. 15.1), 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. Это свойство имеет более высокий приоритет по сравнению с 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", свойства `SourceXml` и `SourceXmlFile` должны быть пустыми.

Трансформационный файл задается свойством

```
property TransformationFile: string;
```

или, если схема трансформации определена документом XML в компоненте TXMLDocument, свойством

```
property TransformationDocument: IDOMDocument;
```

После выполнения трансформации результат сохраняется в свойстве

```
property ResultString: string;
```

в формате XML.

После того, как источник данных и схема трансформации определены, для выполнения преобразования данных, достаточно обратиться к свойству

```
property Data: string;
```

При каждом обращении к нему преобразование выполняется автоматически с помощью вызова метода-обработчика

```
type
  TTranslateEvent = procedure(Sender: TObject; Id: string;
SrcNode: IDOMNode; var Value: string; DestNode: IDOMNode) of object;
property OnTranslate: TTranslateEvent;
```

Например, с диалоговым окном выбора файла XML и клиентским набором данных может быть связан следующий код:

```
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, а 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`.

Приведенный пример с использованием этого метода выглядит так:

```
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 := XMLTrans-
form.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.

15.4. 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. гл. 9)*. Естественно, что для работы с файлами XML необходим специализированный провайдер. 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 пересылает источнику данных сделанные изменения. В качестве источника данных компонент использует файлы в формате XML. Для определения файла предназначено свойство

```
property XMLDataFile: string;
```

Для того чтобы клиентский набор данных (например, компонент `TClientDataSet`) мог использовать данные XML, их необходимо преобразовать в пакет данных Delphi. Обратная операция выполняется при передаче изменений назад.

Вы уже знаете, что таким преобразованиям необходимы трансформационные файлы. В случае с компонентом `TXMLTransformProvider` разработчики Borland поступили просто. Они интегрировали в компонент-провайдер два компонента `TXMLTransform`, которые и преобразуют данные.

Операцию чтения данных из файла XML выполняет один компонент с помощью свойства

```
property TransformRead: TXMLTransform;
```

Операцию записи изменений в файл XML осуществляет другой компонент, используя свойство

```
property TransformWrite: TXMLTransform;
```

Для успешной работы каждому из этих компонентов необходимо придать трансформационный файл. Для этого используется их свойство `TransformationFile`.

Например:

```
XMLTransProvider1.TransformRead.TransformationFile := OpenDlg.FileName;
```

Кроме этого, для каждого из трансформационных компонентов могут быть определены их методы-обработчики, описанные ранее.

Компонент `TXMLTransformProvider` обладает стандартным набором методов-обработчиков компонента-провайдера *(см. гл. 9)*.

На основе компонента `TXMLTransformProvider` можно создавать распределенные приложения различных видов (рис. 15.6).

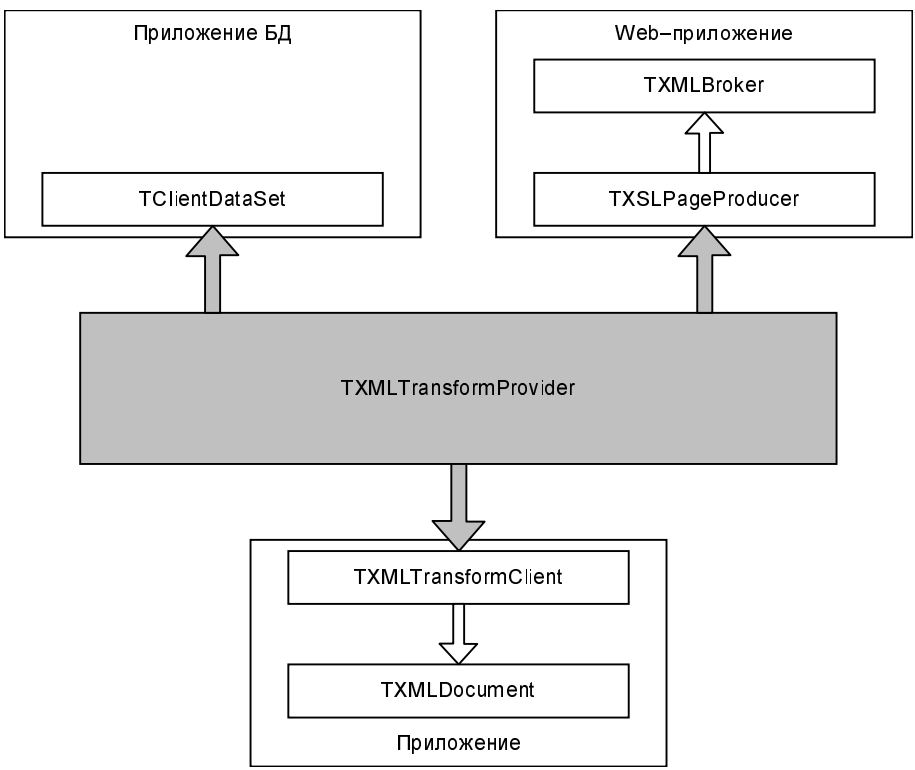


Рис. 15.6. Варианты использования компонента `TXMLTransformProvider` в распределенных приложениях

Во-первых, компонент можно использовать в сочетании 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 компонент-провайдер связывается со страницей HTML мостиком из компонента TXMLBroker.

Рассмотрим эти возможности более подробно.

15.4.1. Данные XML в клиентском наборе данных

Компонент TXMLTransformProvider имеет все необходимые свойства и методы стандартного компонента-провайдера. Следовательно, он может взаимодействовать с компонентом TClientDataSet как локально, так и через удаленный сервер.

Для подключения клиентского набора данных, как и обычно, используется свойство ProviderName, а если источник данных XML расположен на удаленном сервере, то необходимо задать и сам сервер в свойстве RemoteServer *(см. гл. 10)*.

В дальнейшем все операции передачи данных XML клиенту и сохранения изменений выполняются стандартным образом.

Для локального изменения данных используются 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.

15.4.2. Данные XML в документе XML

Помимо традиционного способа (данные XML интегрированы в набор данных) клиентское приложение может использовать данные XML 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 необходимо использовать данные XML на удаленном сервере, нужно дополнительно задать свойство

```
property RemoteServer: TCustomRemoteServer;
```

Для обеспечения взаимодействия с источником данных XML компонент `TXMLTransformClient` содержит три компонента `TXMLTransform`.

Свойство

```
property TransformGetData: TXMLTransform;
```

содержит ссылку на компонент трансформации, обслуживающий входящий поток данных. В зависимости от формата этих данных его свойство `TransformationFile` должно указывать на такой файл трансформации, содержимое которого обеспечит правильное преобразование данных.

Для получения данных от провайдера и преобразования их в формат XML используется метод

```
function GetDataAsXml(const PublishTransformFile: string): string;
```

где параметр `PublishTransformFile` – имя файла трансформации. Но если этот параметр пуст, имя файла будет взято из свойства `TransformGetData.TransformationFile`.

Например, 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, содержимое которого обеспечит правильное преобразование данных.

Если провайдер, связанный с компонентом `TXMLTransformClient`, передает данные из 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 обеспечивает преобразование параметров набора данных.

Параметр `ParamsTransformFile` указывает на файл трансформации, а параметр `ParamsXml` должен ссылаться на документ XML для преобразованных входящих параметров.

15.4.3. Данные XML на странице HTML

Компонент `TXMLTransformProvider` может также использоваться для передачи и преобразования данных для Web-страниц. В этом случае "мостом" между компонентом-провайдером и *компонентом-продюсером* выступает компонент `TXMLBroker` со страницы **Internet Express** Палитры компонентов. 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 разработчик может использовать 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 на `TXMLBroker` должна быть определена в свойстве `XMLData` компонента-продюсера. *Более подробную информацию об этом способе публикации данных с точки зрения создания распределенного Web-приложения см. в гл. 19.*

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 об 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 используется 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 использовать 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, используя метод-обработчик

```
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.

15.5. 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. 15.7). В качестве источника данных используем файл Country.xml из демонстрационной базы данных Delphi, хотя пользовательский интерфейс приложения позволяет выбрать и загрузить любой файл XML.

Компонент `TXMLTransformProvider` обеспечивает доступ к источнику данных. Для этого в его свойство `XMLDataFile` передается полный путь к файлу XML. Кроме этого, компонент-провайдер должен использовать 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 радиокнопок **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 использован компонент `TClientDataSet`, который соединен с провайдером при помощи свойства `ProviderName`. Клиентский набор данных можно просматривать и редактировать в компоненте `TDBGrid`. А сделанные изменения сохраняются с помощью компонента-провайдера. Для этого используется метод

```
ClDataSet.ApplyUpdates(-1);
```

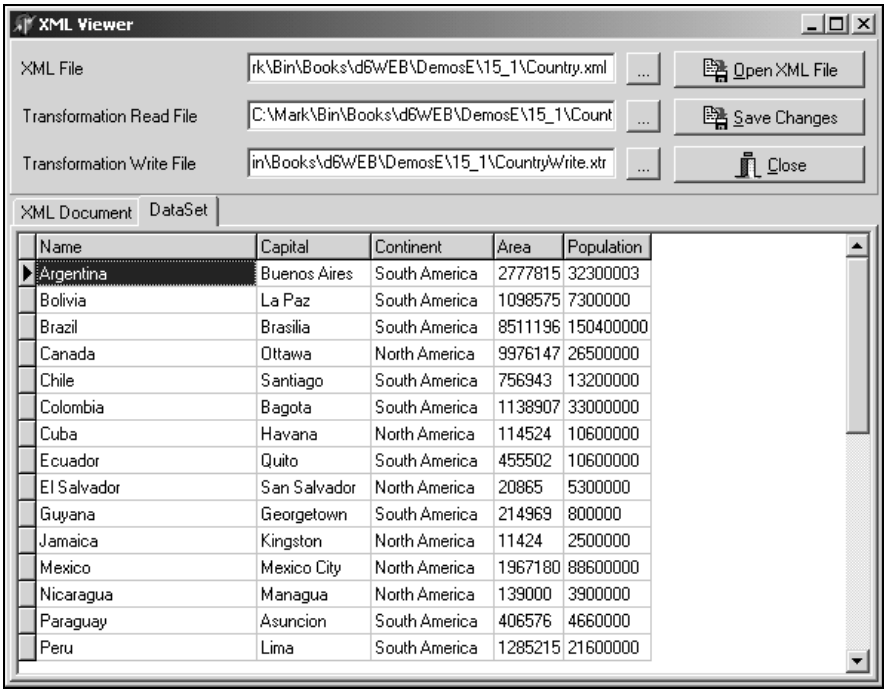


Рис. 15.7. Главное окно приложения XML Viewer

Для представления данных в документе XML использованы компоненты `TXMLTransformClient` и `TXMLDocument`. Первый из них связан с провайдером свойством `ProviderName`. А второй заполняется данными при нажатии на кнопку `bbOpen`. Для этого использован метод `GetDataAsXml` компонента `TXMLTransformClient`:

```
XMLDoc.XML.Clear;
XMLDoc.XML.Add(XMLClient.GetDataAsXml(edReadFileName.Text));
```

Для преобразования данных можно использовать тот же трансформационный файл, что и для чтения данных XML компонента-провайдера.

Затем при помощи метода `SetXMLNode` заполняется дерево узлов и элементов XML в компоненте `TtreeView` (листинг 15.1).

Листинг 15.1. 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 использовать 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 применяются стандартные или минимально модифицированные компоненты. Основную роль здесь играет компонент `TXMLTransformProvider`.

Компоненты преобразования данных XML позволяют передавать данные XML в различные виды клиентских приложений.

- Распределенные приложения баз данных могут работать с данными XML, преобразованными в набор данных, содержащийся в компоненте `TClientDataSet`.
- Любые клиентские приложения могут использовать данные XML, переданные в виде документа XML и представленные компонентами `TXMLDocument` и `TXMLTransformClient`
- Распределенные Internet-приложения имеют возможность публиковать данные XML на Web-страницах.

Для преобразования данных XML в формат пакета данных Delphi (который и позволяет применять эти данные без особого труда) разработчики могут использовать утилиту XML Mapper.

Часть VI

Распределенные приложения и Web-службы

Глава 16. Серверные Web-приложения. Технология Web Broker

Глава 17. Протокол SOAP и Web-службы. Клиентская часть

Глава 18. Серверная часть Web-службы. Совместимость

Глава 19. Технология WebSnap

Глава 16

Серверные Web-приложения. Технология Web Broker

В этой главе мы расскажем о том, как создать приложение, способное взаимодействовать с Web-сервером и публиковать данные в Internet. Под *публикацией* авторы понимают возможность создать и разместить на Web-сервере *динамические* данные, которые, например, содержатся в базе данных, хранятся в виде электронных таблиц или текстовых файлов.

Роль серверного Web-приложения заключается в получении и обработке клиентских запросов и создании ответов на эти запросы. При этом приложение активно взаимодействует с Web-сервером, который и обеспечивает соединение Web-приложения с клиентами.

Приложение может содержать набор HTML-страниц пользовательского интерфейса и должно реализовывать конкретную бизнес-логику. Для этого ему необходимо иметь механизмы обработки и диспетчеризации запросов. Каждый запрос должен быть распознан и направлен для создания ответа соответствующей странице или модулю, инкапсулирующему бизнес-логику приложения.

Для реализации перечисленных возможностей в Delphi включен набор компонентов, модулей и мастеров, объединенных в рамках технологии *Web Broker*. Если вы знакомы с более ранними версиями Delphi (начиная с третьей), вам не следует пугаться этого названия — термин Web Broker теперь используется для обозначения технологии создания Web-приложений начального уровня, той самой, которая вам должна быть уже знакома. Термины Web Services и WebSnap относятся к более новым и сложным технологиям. Однако надо заметить, что они имеют много общего и частично базируются на Web Broker.

При чтении этой главы знание HTML пригодится, но только в объеме, достаточном для увязки непосредственно с вопросами программирования; а художественное оформление вашего продукта лучше оставить Web-дизайнеру.

Наряду с тегами HTML для создания Web-страниц серверных приложений можно использовать и язык XML. Его применение в Web-приложениях мы будем обсуждать в следующих главах.

В этой главе рассматриваются следующие темы:

- интерфейсы CGI и ISAPI;
- из каких частей состоит Web-приложение, и как они взаимодействуют друг с другом;
- компоненты-брокеры и компоненты-продюсеры;
- как использовать язык HTML в приложениях Delphi;
- создание и оформление форм ввода данных;
- компоненты доступа к БД;
- файлы cookies;
- пример создания серверного Web-приложения.

16.1. Публикация данных в Internet. Web-серверы

Механизм публикации данных в Internet подразумевает наличие составных частей, показанных на рис. 16.1. Клиенты при помощи своих приложений (как правило, это обычный Web-браузер) взаимодействуют с Web-серверами, доступными в Internet, посылая им HTTP-запрос с использованием методов `GET` или `POST`. Web-серверы занимаются приемом и диспетчеризацией запросов и передачей ответов клиентам. Для этого запрос должен содержать информацию, достаточную для его передачи запрашиваемому серверному Web-приложению.

Приложение в свою очередь принимает запрос и обрабатывает его, формируя ответ и закладывая в него запрошенную информацию. Естественно, в большинстве случаев данные для ответа формируются динамически. Небольшую базу данных можно "выложить" на десяток или сотню заранее созданных страниц; а как быть с электронным магазином или системой резервирования авиабилетов?

Большая часть Web-серверов работает под управлением UNIX и популярного для этой платформы сервера Apache. Для Windows фирма Microsoft предоставляет сервер Microsoft Internet Information Server (IIS) на платформах Windows NT Server, Windows 2000 и XP (в других ОС фирмы Microsoft есть его упрощенная версия). Кроме этого, вы можете столкнуться с Windows-версией сервера Apache, сервером Netscape FastTrack, а также с OmniPro HTTPD и другими продуктами сторонних разработчиков.

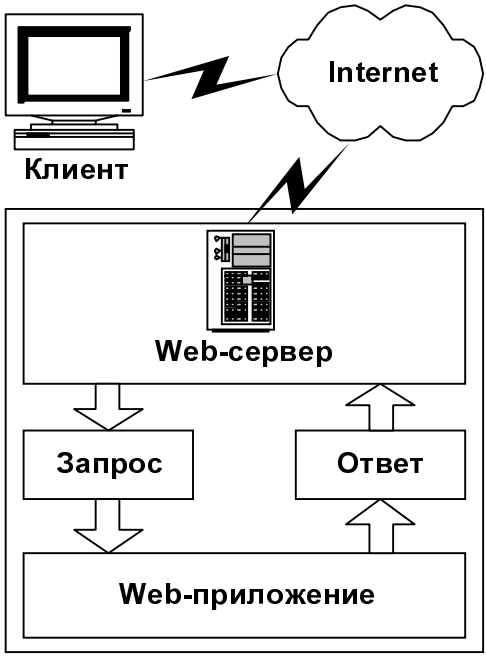


Рис. 16.1. Взаимодействие Web-сервера и серверного Web-приложения

Сервер реализует поддержку протокола HTTP, принимая от браузеров-клиентов запросы и выдавая им в ответ те или иные Web-страницы.

Рассмотрим, например, следующий запрос из браузера-клиента:

http://www.mysite.com/scripts/test.dll/find?name=someuser

Запрошенный URL (Uniform resource locator) выглядит как одна строка, но на самом деле он состоит из нескольких частей.

Примечание

Мы вовсе не хотим пересказывать здесь содержимое документа RFC 1945, описывающего правила кодирования URL. Но некоторые сведения об этих правилах необходимы для Web-разработчика.

Итак, первая часть — адрес хоста: **http://www.mysite.com**. Переходя к терминологии этой главы, хост однозначно указывает на Web-сервер, который будет заниматься обработкой нашего запроса. Далее идет указание на имя серверного приложения (`ScriptName`): в нашем случае это **/scripts/test.dll**. Это означает, что на сервере имеется виртуальная папка scripts (кстати, такая папка создается по умолчанию при установке IIS, и она как раз предназначена для хранения и запуска серверных приложений). В этой папке находится библиотека `test.dll`, которой и адресован запрос.

Вплоть до этого места разбор URL производится Web-сервером; он определяет приложение, вызывает его и передает ему оставшуюся часть URL. Что в ней?

Вслед за именем скрипта идет описание команды, к которой обратился пользователь — эта часть называется `PathInfo`. В нашем случае команда — это строка **find**. Как правило, приложения могут обрабатывать множество команд — иначе для каждой простой операции пришлось бы писать отдельное приложение. В приложениях, написанных на Delphi, командам соответствуют так называемые действия — о них мы будем говорить много и подробно.

Наконец, после вопросительного знака идет собственно текст запроса — набор поименованных параметров, разделяемых символом амперсанда (&). Мы видим в запросе текст **name=someuser**; очевидно, смысл его в поиске в базе данных некоего пользователя `someuser`. Запросы могут иметь поистине гигантскую длину и быть гораздо менее читабельными, чем наш пример. Например, если запрос содержит текст на одном из национальных языков (в т. ч. русском), он передается в шестнадцатеричной кодировке. Тем не менее, синтаксис и принцип работы запросов такой же.

Непосредственной обработкой запросов и формированием ответов занимаются серверные приложения. Ясно, что любой Web-сервер должен поддерживать возможности расширения, а именно — протокол для вызова и передачи данных серверным приложениям. Кратко перечислим основные варианты.

С сервером IIS очень часто используется технология Active Server pages (ASP, Активные серверные страницы), специально созданная для него фирмой Microsoft. Такие страницы представляют собой код на языке *сценариев* или *скриптов* — VBScript или JScript.

Примечание

В фирме Microsoft табуировано слово Java, и она называет JScript то, что другие производители называют JavaScript. Более того, диалекты этого языка отличаются между собой, что сильно затрудняет кросс-платформенную разработку. Вы также можете встретить название ECMA-script. Так называется совместимое и стандартизованное подмножество JavaScript.

Кроме кода, в теле ASP могут находиться теги HTML — таким образом, страница может сочетать в себе статическую (HTML) и динамическую (код сценария) часть.

В принципе, технология ASP хорошо развита, отработана и документирована. Она представляет собой весьма серьезную альтернативу Web Broker. Однако принципиальное отличие состоит в том, что код сценариев является интерпретируемым; серверные приложения, написанные на Delphi, скомпилированы в виде исполняемых модулей (EXE или DLL) и выполняются быстрее.

Однако это не значит, что мы ставим крест на языках сценариев. В рамках технологии Web Broker (а также некоторых других технологий, описываемых далее в этой книге) можно применять сценарии, созданные с использованием языков VBScript и JScript.

А теперь перейдем к вариантам, реализованным непосредственно в Delphi — протоколам CGI и ISAPI.

16.2. Введение в интерфейсы CGI и ISAPI

Первым (и общепринятым) интерфейсом для создания модулей расширения Web-серверов был CGI (Common Gateway Interface), пришедший в Windows из UNIX. Web-приложение в этом случае представляет собой сценарий или обычный исполняемый файл (EXE), получающий от сервера необходимую информацию из переменных окружения. В Windows эти особенности реализации являются неприятными — 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 означает существенную потерю 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. Кроме того, сервер указывает приложению имя временного файла, в который оно должно поместить результаты выполнения. Дождавшись окончания работы приложения, сервер отправляет клиенту сгенерированный файл.

Оба интерфейса относятся к числу простых, но не слишком удобных для работы в Windows. Нужно сказать, что CGI — это лишь формат передачи параметров от сервера к приложению; CGI может использоваться как сценариями, так и традиционными exe-файлами. Излюбленным средством написания сценариев CGI является специальный язык Perl; кроме него такие приложения можно писать и на C, Basic, и, естественно, в среде Delphi.

Но есть принципиально иной путь ускорения работы Web-приложений. Microsoft снабдила свой сервер Internet-интерфейсом ISAPI (Internet Services API). Он обеспечивает поддержку приложений на основе динамических библиотек. Выгоды этого очевидны — загрузка DLL занимает гораздо меньше времени, чем создание процесса; к тому же, Web-сервер обычно создает пул потоков, загружающих DLL, и кэширует их; так что временные затраты минимальны.

Разновидность интерфейса NSAPI (Netscape Server API) обеспечивает работоспособность приложений при работе с Web-сервером Netscape.

Стандартом ISAPI предусмотрено создание двух типов приложений — фильтров и расширений. ISAPI-фильтры могут перехватывать HTTP-трафик

с целью, например, расшифровки, сжатия/восстановления, протоколирования, проверки подлинности и т. п. Обработанные данные могут передаваться другому приложению. Такие задачи довольно специфичны; Delphi не содержит специальных средств для написания ISAPI-фильтров, хотя в принципе это возможно. Не будем их рассматривать и мы.

Гораздо более распространены *ISAPI-расширения* (ISAPI extensions) — "конечные точки" маршрутов, по которым идут запросы пользователей.

Такие динамические библиотеки работают в адресном пространстве сервера Internet Information Server или в специально создаваемом процессе (процессах). Для сервера IIS версии 5 существуют три разновидности запуска ISAPI-расширений, зависящие от установленной степени защиты. Параметры защиты приложений вы можете увидеть на странице "Домашний каталог" вашего Web-узла.

- Если установлена низкая степень защиты, то ISAPI DLL загружается и исполняется в том же адресном пространстве, что и сам Web-сервер (в процессе под названием inetinfo.exe). Это — наиболее быстрый, но и наименее безопасный путь. Любая ошибка в коде вашего ISAPI-приложения может привести к повреждению и неработоспособности всего сервера.
- При средней степени защиты (это значение принимается по умолчанию) ISAPI-расширения загружаются и исполняются специальным дочерним процессом dllhost.exe, который, в свою очередь, порожден сервером.
- При высокой степени защиты для каждого ISAPI-модуля запускается отдельный процесс dllhost.exe, и тем самым достигается полная изоляция модуля от всех прочих приложений.

Модуль расширения ISAPI/NSAPI должен экспортировать вполне определенные функции, которые и будут вызываться Web-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` информацию о версии приложения ISAPI или NSAPI.

Функция

```
function HttpExtensionProc(var ECB: TEXTENSION_CONTROL_BLOCK): DWORD;
```

обеспечивает передачу запроса от Web-сервера приложению; параметр ECB (так называемый Extension Control Block) содержит всю необходимую информацию.

И функция

```
function TerminateExtension(dwFlags: DWORD): BOOL;
```

возвращает `True`, если приложение готово к закрытию.

Три эти функции обеспечивают низкоуровневый интерфейс — непосредственно взаимодействие в стандарте ISAPI. Но... в 99% случаев они вам не понадобятся, поскольку в Delphi над ними существует удобная и простая объектная надстройка. Название этой надстройки — Web Broker, к ее рассмотрению мы и приступаем.

16.3. Типы серверных Web-приложений

Давайте посмотрим, какие именно типы приложений доступны для разработки в Delphi.

Для создания серверного Web-приложения необходимо в Репозитории Delphi перейти на страницу **New** и выбрать значок **WebServer Application**. После этого появляется диалоговое окно, в котором необходимо выбрать тип приложения (рис. 16.2).

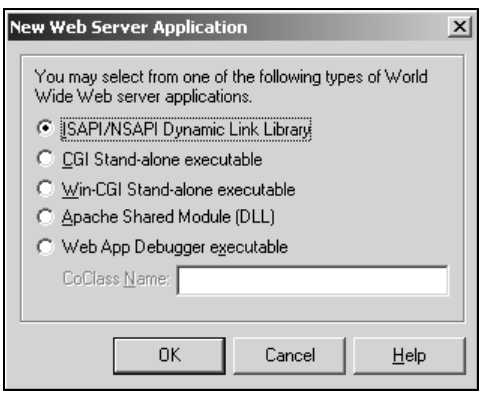


Рис. 16.2. Диалоговое окно выбора типа серверного Web-приложения

В группе радиокнопок доступны следующие типы приложений.

- **ISAPI/NSAPI Dynamic Link Library** — приложение на основе динамической библиотеки. Для обмена данными использует интерфейс ISAPI.
- **CGI Stand-alone executable** — приложение на основе исполняемого файла. Для обмена данными использует интерфейс CGI, получающий управляющие команды через переменные окружения приложения. Для каждого запроса создается отдельный экземпляр приложения.

- **Win-CGI Stand-alone executable** — разновидность приложения CGI, адаптированная для использования в Windows. Отличие состоит в том, что параметры запроса передаются в INI-файле.
- **Apache Shared Module (DLL)** — приложение на основе динамической библиотеки, работающей в адресном пространстве Windows-версии сервера Apache в многопользовательском режиме. Для каждого запроса создается собственная нить.
- **Web App Debugger executable** — приложение на основе исполняемого файла. Этот тип приложения взаимодействует с утилитой Web App Debugger, которую можно запустить командой пункта меню **Tools\Web App Debugger** главного окна Delphi. Эта утилита представляет собой модель Web-сервера и заменяет настоящие Web-серверы. С ее помощью можно ускорить процесс отладки серверных Web-приложений средствами Delphi без использования реальных Web-серверов. Затем готовое приложение легко переносится в среду взаимодействия с реальным сервером.

Не пугайтесь обилия возможностей. Все эти типы приложений отличаются только способом взаимодействия с сервером; а поскольку взаимодействие ложится на плечи объектов Web Broker, то на вашем пользовательском коде смена типа приложения практически никак не скажется. Вам достаточно перенести созданные модули в приложение требуемого типа. Давайте поговорим об этом подробнее после создания наших тестовых примеров.

Итак, если вам важны соображения переносимости, то обратите внимание на приложение CGI (этот стандарт поддерживается буквально всеми серверами) или модуль Apache (он входит в состав и широко применяется в Linux); такие приложения вы сможете компилировать как в Delphi, так и в Borland Kylix. Приложения WinCGI — это, похоже, отмирающая технология; рекомендовать их вам мы не будем.

Поскольку мы с вами работаем в среде Windows, то далее в этой главе сосредоточимся на серверных приложениях ISAPI как современном и быстром средстве создания приложений для Microsoft Internet Information Server и совместимых с ним Web-серверов.

16.4. Структура серверного Web-приложения в Delphi

Итак, серверное приложение взаимодействует с Web-сервером, получая от него запросы и возвращая ответы на них. Посмотрим, что же происходит при этом внутри приложения и какие компоненты Web Broker в нем используются.

С точки зрения функциональности, серверное Web-приложение состоит из трех взаимосвязанных частей, выполняющих определенные операции (рис. 16.3).

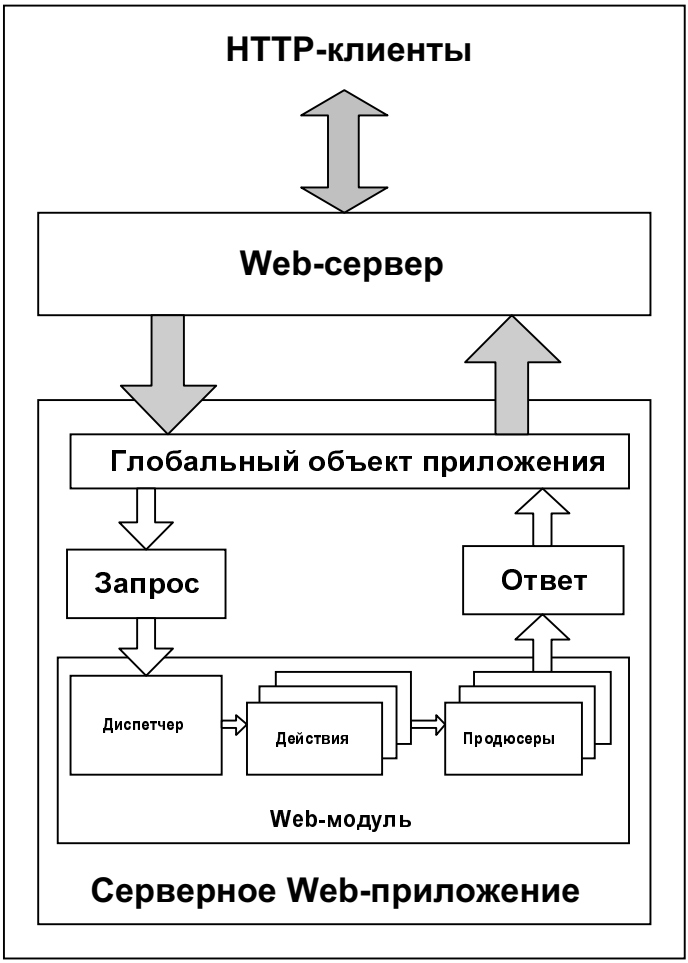


Рис. 16.3. Структура серверного Web-приложения

Непосредственное взаимодействие с сервером осуществляет *глобальный объект приложения*. Его роль — создание и инициализация всех структур, необходимых для запуска вашего кода, содержащегося в Web-модуле. В ряде случаев этот глобальный объект также управляет пулом потоков, обрабатывающих запросы, — это необходимо для приложений, работающих в режиме высокой нагрузки.

С точки зрения программиста, этот глобальный объект приложения нивелирует разницу между перечисленными ранее типами интерфейсов — ISAPI, CGI, Apache и т. п.

Процесс управления запросами называют *диспетчеризацией запросов*. Приложение должно уметь обработать запрос и получить из него всю необходимую

информацию, а именно команду (`PathInfo`) и ее параметры. Например, если ваше приложение обрабатывает команды манипулирования данными в СУБД, то все параметры для построения SQL-команды содержатся в строке HTTP-запроса. Но если компонентов для работы с SQL в проекте много, какому из них адресован запрос? Функции диспетчеризации исполняются специализированным компонентом `TWebDispatcher`.

Диспетчер выполняет четыре основные операции:

- по поступившей к нему строке запроса определяет вызываемую команду;
- формирует из поступившей от сервера информации и списка параметров объект `TWebRequest`;
- вызывает так называемое действие — код, отвечающий за обработку данной команды;
- получает объект ответа `TWebResponse`, при необходимости дополняет и модифицирует его и отдает вызвавшему приложению.

Сам по себе компонент `TWebDispatcher` применяется не очень часто, но он является составной частью Web-модуля — компонента `TWebModule`.

Web-модуль — объект, объединяющий в себе роли модуля данных и Web-диспетчера. Это следует понимать буквально: `TWebModule = TDataModule + + TWebDispatcher`. Поэтому Web-модуль, с одной стороны, инкапсулирует действия, а с другой — прекрасно справляется и с традиционной ролью модуля данных, а именно служит площадкой для хранения невизуальных компонентов и их программного кода. В рамках технологии Web Broker в приложении может быть один и только один компонент-диспетчер, поэтому, в отличие от модулей данных `TDataModule`, вы не можете иметь более одного `TWebModule` или поместить на `TWebModule` компонент `TWebDispatcher`.

Реализация бизнес-логики — это вызов приложением в ответ на поступивший и правильно обработанный запрос некой заранее созданной и отлаженной процедуры, выполняющей нужную клиенту операцию. Ею 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 используются специальные объекты-*действия (Actions)*. Действия имеют методы-обработчики события `OnAction`, которые вызываются диспетчером при обработке запроса. Именно в них вам надлежит реализовывать необходимые приложению операции. Коллекция действий находится в каждом диспетчере.

Альтернативным способом реализации бизнес-логики является использование сценариев (скриптов) на языках VBScript или JScript. Эти скрипты размещаются на Web-страницах и вызываются при необходимости.

Имея в виду, что пользовательский интерфейс клиента — это в большинстве случаев браузер, серверное приложение обычно возвращает *Web-страницы* —

набор тегов HTML (или XML). Разумеется, вы можете средствами Delphi просто выбрать готовую — статическую — страницу и вернуть ее пользователю. Однако с такой задачей справляется и непосредственно Web-сервер, и программирование тут ни при чем. Предпочтителен *динамический способ формирования страниц*; для этого в Delphi используются *компоненты-продюсеры*, которые умеют формировать страницу на основе *шаблонов* HTML или XML, а также таблиц баз данных и SQL-запросов; то есть они создают набор тегов HTML и код сценариев, обеспечивающих корректное представление и поведение страницы в браузере клиента. На странице **Internet** Палитры компонентов доступны несколько компонентов-продюсеров для доступа к разным источникам данных.

16.4.1. Глобальный объект приложения

Исходя из структуры и схемы взаимодействия приложения с Web-сервером (см. рис. 16.1 и 16.3), очевидно, что должен существовать программный код, обеспечивающий получение запросов и отправку ответов, и общее управление частями приложения. Для выполнения этих задач каждое приложение, созданное с использованием технологии Web Broker, содержит глобальный объект приложения.

Этот объект включается в состав приложения автоматически при его создании (вместе с модулем `WebBroker`) и инициализируется при запуске приложения. Доступ к нему из вашего кода осуществляется при помощи глобальной переменной `Application`, подобно обычным приложениям Delphi, в которых доступ к экземпляру класса TApplication выполняется также с помощью глобальной переменной. На самом деле это одноименная глобальная переменная с совсем другими свойствами и методами. Поэтому ни при каких обстоятельствах не включайте в проект вашего Web-приложения модуль `Forms` или другие модули, ссылающиеся на него; в этом случае неизбежен конфликт и неправильная инициализация `Application`.

Для каждого типа серверного Web-приложения из перечисленных выше имеется собственный класс, а их общим предком стал класс `TWebApplication` (рис. 16.4).

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.

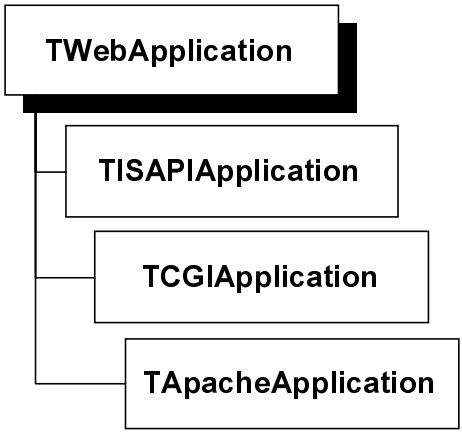


Рис. 16.4. Иерархия классов серверных Web-приложений

Создание и инициализация глобального объекта зависит от типа приложения, описанного в *разд. 16.3 данной главы*. После создания методы объекта вызываются Web-сервером по мере надобности. Чаще всего — это направление запроса приложению.

Рассмотрим работу глобального объекта на основе приложения ISAPI.

Примечание

Схема работы приложения Apache совпадает с ISAPI, а приложения CGI можно приближенно рассматривать как упрощенный вариант ISAPI с единичным запросом. Рассмотренные ниже свойства глобального объекта соответствуют приложению ISAPI.

После получения запроса глобальный объект приложения формирует 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 передается диспетчеру (это может быть Web-модуль или компонент `TWebDispatcher`). И в конце "технологической цепочки" готовый ответ направляется Web-серверу (если это еще не сделано объектом действия), а экземпляр Web-модуля уничтожается или кэшируется для дальнейшего использования.

Дополнительно глобальный объект позволяет получать информацию о текущей сессии приложения.

Свойство, доступное только для чтения

```
property ActiveCount: Integer;
```

возвращает число запросов, обрабатываемых в настоящее время.

Если предполагается использовать Web-модули в пуле, без удаления, то свойству

```
property CacheConnections: Boolean;
```

необходимо присвоить значение `True`. Если пулинг Web-модулей применяется, 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 числа `MaxConnections`. При попытке создать Web-модуль с количеством запросов свыше `MaxConnections` генерируется исключительная ситуация.

16.4.2. Web-модуль

Основой проекта ISAPI является Web-модуль, который представлен классом `TWebModule`. На нем, как на фундаменте, стоит любое из описываемых в этой главе приложений. Он интерпретирует запросы и направляет их для выполнения.

Для этого класс Web-модуля наследует функциональность базового диспетчера запросов — класса `TCustomWebDispatcher`. Аналогичными возможностями обладает и компонент `TWebDispatcher`. Благодаря этому разработчики могут создавать приложения без использования Web-модулей; скажем, вы можете соединить `TWebDispatcher` с модулем данных уже существующего приложения.

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. Каждый из них идентифицируется текстовой строкой — свойством `TWebActionItem.PathInfo`. Вспомним структуру URL. В ответ на запрос:

```
http://www.mysite.com/scripts/test.dll/add
```

будет вызвано действие, свойство `PathInfo` которого равно `add`.

Кроме того, протоколом HTTP установлено несколько методов взаимодействия между клиентом и сервером — например, типичные `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`.

Эти два свойства и определяют диспетчеризацию. Если при помощи нужного метода прислан запрос с правильным `PathInfo`, то вызывается соответствующее действие.

Обработке запросов, для которых не найдено соответствующее действие, Web-модуль может назначить действие по умолчанию. Это действие, значение свойства `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. Поскольку во время создания или извлечения из пула Web-модуль получает ссылки на объекты запроса и ответа, сформированные глобальным объектом приложения, 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 используются 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 в момент создания и уничтожения 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.

Отдельно нужно остановиться на использовании системного реестра. Дело в том, что модули, исполняемые Web-сервером, как правило, функционируют в режиме системных служб (когда может не быть интерактивного пользователя, осуществившего вход в систему) с правами и в контексте безопасности анонимного пользователя `IUSR_<MACHINENAME>`. При этом ветви реестра `HKEY_CURRENT_USER` нет (так как нет ни самого текущего пользователя,

ни его профиля, в котором хранится ветвь HKEY_CURRENT_USER). К ветви HKEY_LOCAL_MACHINE доступ часто ограничен из соображений безопасности, особенно это касается упомянутого анонимного пользователя, являющегося членом группы guests.

Поэтому если ваше приложение имеет любые настраиваемые параметры, то вам нужно будет запоминать их не в реестре, а в соответствующем INI-файле.

16.4.3. Действия

Функциональность отдельного действия реализована в свойствах и методах класса TWebActionItem.

Во-первых, 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 по умолчанию, свойство

```
property Default: Boolean;
```

которого имеет значение True.

Процесс поиска действия и выполнения его метода-обрабочика OnAction выполняется Web-модулем автоматически.

При необходимости действие можно включить и отключить. Для этого используется свойство

```
property Enabled: Boolean;
```

Если оно имеет значение False, диспетчер запросов пропускает действие при поиске кандидата для обработки запроса.

16.4.4. Запросы и ответы

При поступлении запроса от Web-сервера глобальный объект приложения создает объекты запроса и ответа.

Класс `TWebRequest` инкапсулирует запрос. Класс `TWebResponse` описывает 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 удобно использовать свойство

```
property QueryFields: TStrings;
```

которое представляет собой список строк, каждая из которых содержит один параметр запроса в формате `Name=Value`.

Также объект запроса возвращает 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 у них значения 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.

16.4.5. Компоненты-продюсеры

В состав технологии Web Broker входят компоненты-продюсеры. Они обеспечивают динамическую генерацию Web-страниц на основе шаблонов HTML или используют базы данных в качестве источника данных.

Наиболее общие функции продюсера реализованы в компоненте `TPageProducer`.

Шаблон для создания страницы задается свойством

```
property HTMLDoc: TStrings;
```

Если шаблон хранится в файле, то его можно определить с помощью свойства

```
property HTMLFile: TFileName;
```

Для генерации страницы на основе шаблона необходимо вызвать метод

```
function Content: string;
```

но он выполнится, только если шаблон задан перечисленными выше свойствами. В противном случае можно использовать 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 `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. 16.1).

Таблица 16.1. *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.

16.5. 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 есть широчайший выбор специальной литературы. И в то же время рассказ о страницах с приветствиями вроде "Hello, World!" вряд ли окажется полезным. Мы уделим внимание только некоторым тонкостям использования компонентов Web Broker, которые помогут в реальной разработке HTML-страниц.

16.5.1. 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 страницы будут выглядеть похоже. Таким образом, можно провести настройку внешнего вида, прочитав часть параметров из файла инициализации (INI) или реестра (листинг 16.1).

Листинг 16.1. 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 — применить компоненты-продюсеры Web Broker.

Примечание

Данные компоненты могут быть использованы не только в составе ISAPI- или CGI-приложения, но и в любом другом 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.

16.5.2. Ввод и редактирование данных

Мы познакомились со способом отображения информации на Web-страницах. Но это только половина дела — нужно еще уметь ввести данные 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 служат теги `<INPUT>`. Они специально предназначены для организации диалога с пользователем в среде браузера и имитируют основные элементы управления Windows, предназначенные для ввода информации. В каждом теге должно быть поле `type` (задающее тип элемента управления) и поле `name`. Основные типы элементов управления приведены в табл. 16.2.

Таблица 16.2. *Типы элементов управления Web-страниц (тег `<INPUT>`)*

Тег	Назначение
`<input type=TEXT>`	Однострочный текстовый редактор 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 начальное значение

Теперь создадим шаблон страницы, содержащий набор элементов управления для ввода имени пользователя и пароля (листинг 16.2).

Листинг 16.2. 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 два для обязательных кнопок — **Submit** и **Reset**, аналогов **OK** и **Cancel** в Windows. Кстати, текст на этих кнопках может быть изменен заданием поля `value`.

В итоге страница должна выглядеть, как показано на рис. 16.5.

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 код Object Pascal должен обеспечивать проверку данных, введенных пользователем. Метод-обработчик действия `actValidate` для этого запроса может выглядеть, как приведенный в листинге 16.3.

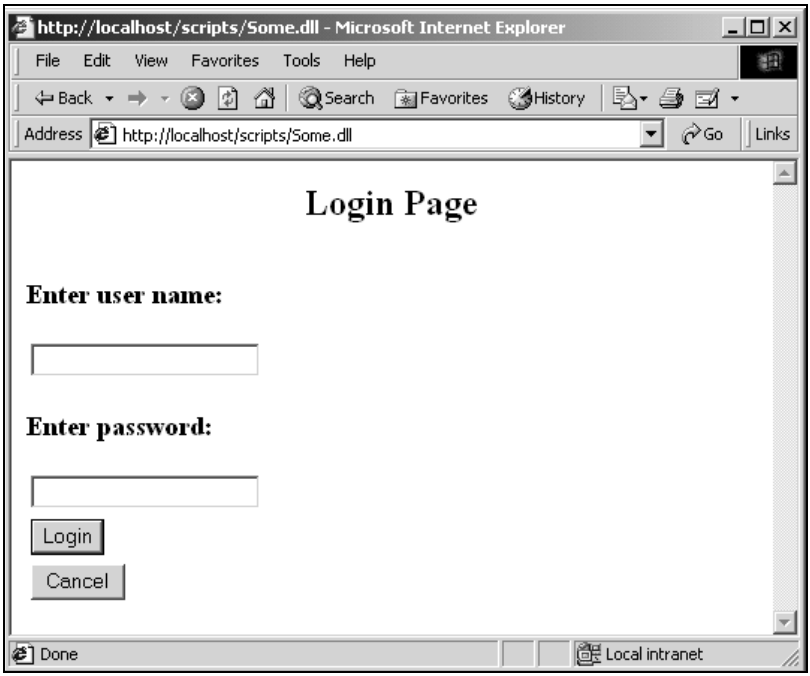


Рис. 16.5. Внешний вид страницы аутентификации пользователей

Листинг 16.3. 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` устанавливается наличие введенных данных 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.

16.6. Cookies

Серверное Web-приложение может отвечать на запросы анонимных клиентов, но гораздо удобнее, если оно "помнит" хотя бы что-нибудь о своих прошлых отношениях с клиентом — например, 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 ключиков — *cookies*.

Cookies работают следующим образом. Когда приложение идентифицировало клиента, оно может добавить к своему HTTP-ответу специальное 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 к 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;
```

16.7. 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.

- Компонент `TDataSetPageProducer` создает HTML-текст страницы, используя в качестве основы набор данных и шаблон HTML.
- Компонент `TDataSetTableProducer` генерирует HTML-текст страницы, представляя данные из любого набора данных в табличном виде.
- Компонент `TQueryTableProducer` генерирует HTML-текст страницы, представляя данные из набора данных, созданного параметризованным компонентом запроса SQL в табличном виде. При этом могут использоваться только компоненты запросов BDE.
- Компонент `TSQLQueryTableProducer` генерирует HTML-текст страницы, представляя данные из набора данных, созданного параметризованным компонентом запроса SQL в табличном виде. При этом могут использоваться только компоненты запросов dbExpress.

Рассмотрим их подробнее.

16.7.1. Публикация записей таблиц баз данных

Для того чтобы опубликовать на создаваемой странице не просто информацию, но информацию из базы данных, необходимо использовать компонент `TDatasetPageProducer`. Его свойства и методы в большинстве своем совпадают со свойствами и методами обычного компонента-продюсера.

Дополнительно, для связывания с набором данных он имеет свойство

```
property DataSet: TDataSet;
```

из которого можно получить ссылку на любой компонент доступа к данным *(см. часть 2 "Технологии доступа к данным")*. Для создания Web-страницы с данными из соответствующей таблицы базы данных заполнения только этого свойства недостаточно.

Кроме этого, `TdatasetPageProducer`, как и стандартный компонент-продюсер, должен иметь шаблон HTML. Шаблон предоставляет собой обычный документ или текст HTML для Web-страницы, но в нем в необходимых местах должны быть включены теги, имена которых совпадают с именами полей обрабатываемого 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:

```
<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 заметили, 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 пойдет речь дальше.

16.7.2. Генерация отчетов

При публикации данных в Internet сложнее всего генерировать отчеты и выходные формы. 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 роль своеобразного генератора табличных отчетов на Web-страницах.

> **Примечание**
>
> На самом деле компонентов три, но компоненты `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 настраивать 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, используя текст 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 — он в большинстве случаев работает быстрее.

Документ, генерируемый при помощи TDatasetTableProducer и TQueryTableProducer, состоит из четырех частей.

1. Заголовок документа. Он 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. Для того 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 содержатся в свойстве

```
property Footer: TStrings;
```

Имеет смысл для работы с таблицей 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. 16.6) и добавляя новые столбцы, мы можем связать их с полями источника данных, указанного в свойстве `DataSet` (или `Query`).

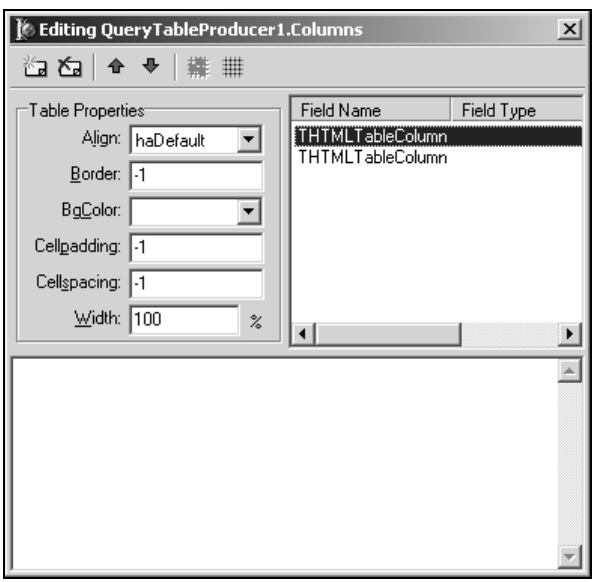


Рис. 16.6. Редактор свойства `Columns` компонентов `TDatasetTableProducer` и `TQueryTableProducer`

Кроме того, 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. Значения перечисленных свойств — `BgColor`, `Align`, `vAlign` — напрямую соответствуют тому, что будет содержаться в этих тегах.

В левой части редактора свойства `Columns` (см. рис. 16.6) можно поменять значения атрибутов всей таблицы, которые входят в состав свойства

```
property TableAttributes: THTMLTableAttributes;
```

Эти свойства имеют следующий смысл (табл. 16.3).

Таблица 16.3. *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

Тем, кто знает HTML, понятно, что значения из `TableAttributes` соответствуют модификаторам тега `<TABLE>`.

Можно отдельно задать и оформление строк — для этого предусмотрено свойство:

```
property RowAttributes: THTMLTableRowAttributes;
```

Для того чтобы отчет не получился слишком длинным (а Web-страница — слишком большой и неудобной для загрузки), можно ограничить число строк (то есть 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 его генерацию.

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, вот пример:

```
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.

16.7.3. Редактирование данных

И в завершение разговора об использовании баз данных рассмотрим возможности редактирования данных в Delphi. Для этого перейдем к более сложному варианту организации ввода данных, который реализуется в простой задаче формирования курсора 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.

```
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 нее. Для этого в последний столбец добавляется по одному флажку (с именами R1...RN) и текстовому полю (с именами T1..TN). Если пользователь что-то выбрал, он должен установить флажок и ввести число в соседнем текстовом 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>`.

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 установлены пользователем. И если флажок R1 установлен, 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 производится. 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.

16.8. Пример простого серверного Web-приложения

И в заключение рассмотрим небольшой пример. Целью его создания является не демонстрация использования компонентов Web Broker (достаточно подробно об этом уже рассказывалось), а необходимость обсудить процесс разработки приложения в целом и методы его отладки. Поэтому пример отличается примерной простотой.

При помощи Репозитория Delphi создадим новое серверное Web-приложение. В диалоговом окне (см. рис. 16.2) 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.

16.8.1. Главное окно приложения

После создания проекта в среде разработки доступны два модуля. Unit1.pas представляет обычную форму и автоматически будет использован для работы утилиты Web App Debugger. Нас же интересует Unit2.pas, который содержит Web-модуль. Переименуем его в `wmMain`.

Перенесем в Web-модуль со страницы **Internet** Палитры компонентов компонент `TPageProducer` и назовем его `MainPage`. Его свойство `HTMLDoc` содержит шаблон главной страницы (листинг 16.4). На ней доступны две гиперссылки — на две страницы, отображающие в виде списка и таблицы данные из таблицы `Country` демонстрационной базы данных Delphi.

Листинг 16.4. 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 16.4 видно, что в шаблоне присутствуют теги `<#APPNAME>`. В методе-обработчике `OnHTMLTag` компонента `MainPage` эти теги заменяются на имя приложения, формируя корректную гиперссылку (см. листинг 19.5).

Для того чтобы страница `MainPage` открывалась при запуске приложения, в Web-модуле необходимо создать действие (используя свойство модуля `Actions`), связать его с компонентом `TPageProducer` (свойство `Producer`

созданного действия) и сделать его используемым по умолчанию (используя свойство `Default` созданного 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 с помощью действия `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 19.4), необходимо добавить в Web-модуль еще два действия — `actCountryList` и `actCountryTable`. Они должны быть связаны с соответствующими компонентами-продюсерами и их свойства `PathInfo` должны совпадать с запросами для гиперссылок в шаблоне главной страницы.

16.8.2. Просмотр таблицы *Country*

Теперь займемся таблицей `Country`. Перенесем со страницы **ADO** в Web-модуль компонент `TADOTable` и настроим его на таблицу `Country` демонстрационной базы данных Delphi (листинг 16.5). Для этого используем файл ..\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb базы данных Microsoft Access. *Более подробную информацию о технологии ADO см. в гл. 7.*

Затем со страницы **Internet** Палитры компонентов перенесем в Web-модуль компоненты `TDataSetPageProducer` и `TDataSetTableProducer`.

Листинг 16.5. 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 в виде списка данных из таблицы. 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.

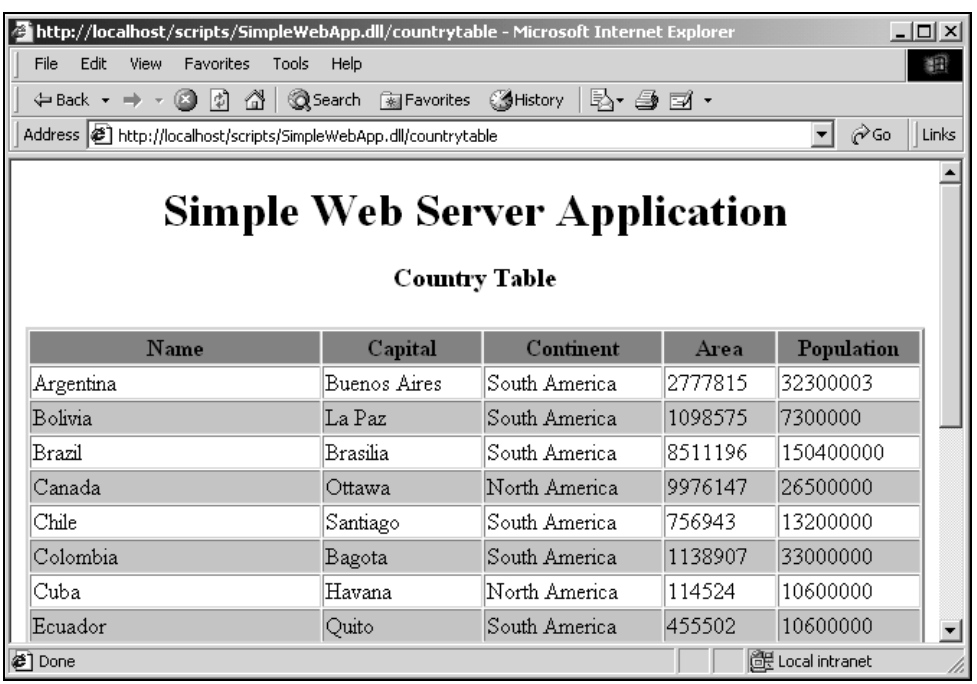


Рис. 16.7. Страница приложения SimpleWebApp.exe

Шаблон содержит тег <#COUNTRYLIST>, который в методе-обработчике OnHTMLTag компонента заменяется на данные, полученные из набора данных Country (см. листинг 19.5).

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 доработки 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 представит данные, как показано на рис. 16.7.

Примечание

Обратите внимание, что для заполнения обеих страниц используется единственный компонент набора данных.

16.8.3. Тестирование приложения

На этом создание приложения закончено. После сохранения и компиляции его можно запустить при помощи утилиты Web App Debugger (рис. 16.8).

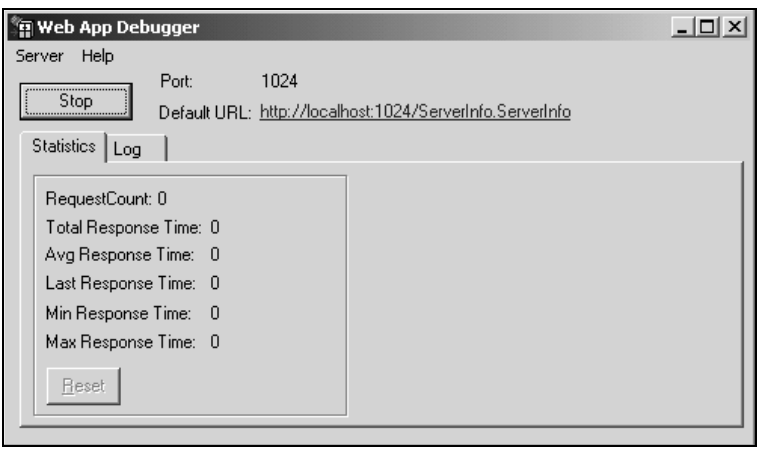


Рис. 16.8. Окно утилиты Web App Debugger, позволяющей отлаживать серверные Web-приложения без реальных Web-серверов

Кнопка **Start** | **Stop** в верхней части окна позволяет обновить список зарегистрированных приложений утилиты 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.

16.8.4. Перенос приложения на платформу ISAPI

Использование типа приложения Web App Debugger executable позволяет существенно ускорить процесс отладки. Но для распространения этот тип, конечно же, не подходит. Поэтому опишем простой процесс переноса приложения на платформу ISAPI.

1. Создайте новый проект серверного Web-приложения и сохраните его в отдельной папке, используя те же имена 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.

Есть и другой путь — включить созданный Web-модуль в состав Репозитория объектов Delphi и затем повторно использовать его в других типах приложений.

Отладка ISAPI- приложений

Как и любое другое приложение, динамические библиотеки ISAPI нуждаются в отладке. Но это — не обычные 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 — так как 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 ее кэш приложений сбрасывается.

   Процедура эта не быстрая 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 сулит гигантское снижение производительности сервера; не забудьте снова включить кэширование, когда все ваши приложения будут отлажены.

Теперь непосредственно об отладке. "Напрямую" отлаживать модули ISAPI нельзя, так как это — динамические библиотеки, и они нуждаются в запускающем приложении. В Delphi 6 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.

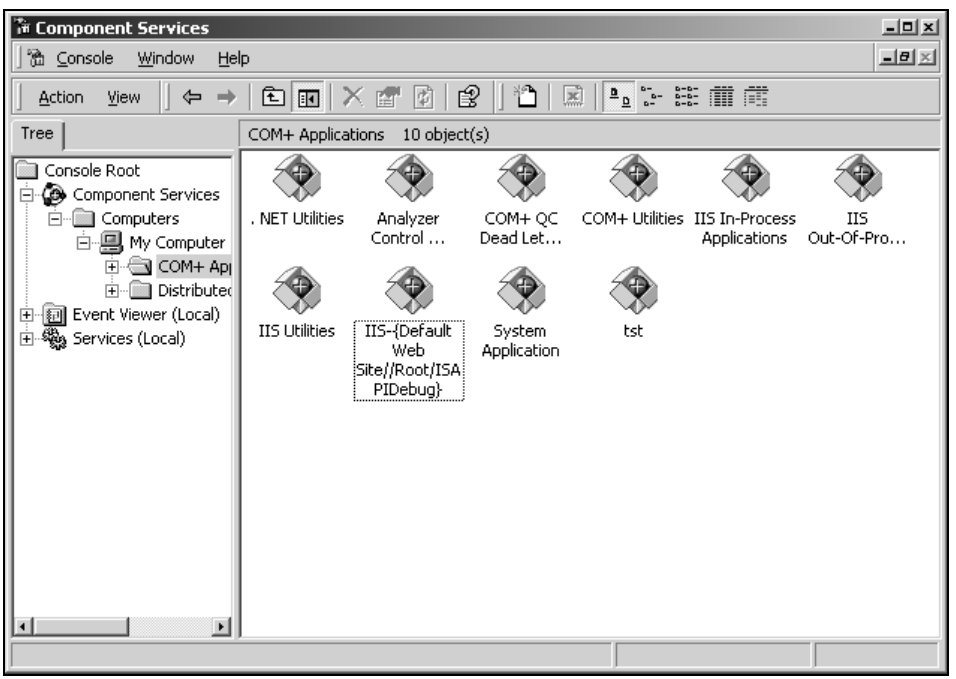
Более изящный подход изложен на сайте **www.msdelphi.com** в статье "Debugging IIS5 the easy way". Отлаживаясь в процессе IIS, вы подвергаете его риску и дополнительным нагрузкам. Описываемый ниже подход лишен этого недостатка.

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. 16.9, *а* оно называется **IIS-{Default Web Site//Root/ISAPIDebug}**.
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. 16.9, *б* — **Application ID**, равный в нашем

случае **{DB431D8F-292D-4EC8-BD37-054C1BB6378D}**). Во-вторых, следует изменить настройки идентификации по умолчанию. На странице **Identity** этого диалогового окна 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. 16.9, *в*).

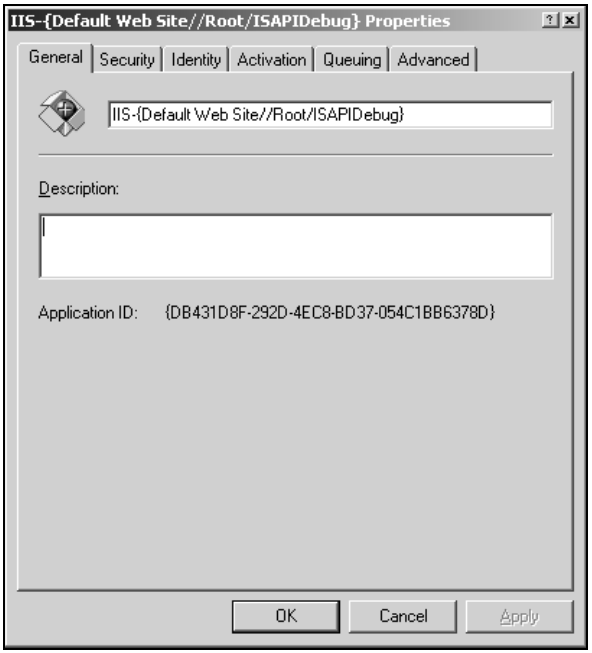
3. На этом настройка dllhost закончена. Для установки параметров в Delphi нам понадобится `Application ID`, который к этому моменту должен быть у вас в буфере обмена. Параметры запуска вашего приложения в Delphi выглядят, как показано на рис. 16.9, *г*; разумеется, если путь к системным папкам у вас отличается от указанного, замените значение параметра **Host Application**.

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 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 http://localhost/ISAPIDebug/SimpleWebApp.dll. Вот тогда-то среда Delphi и сможет получить управление для целей отладки.

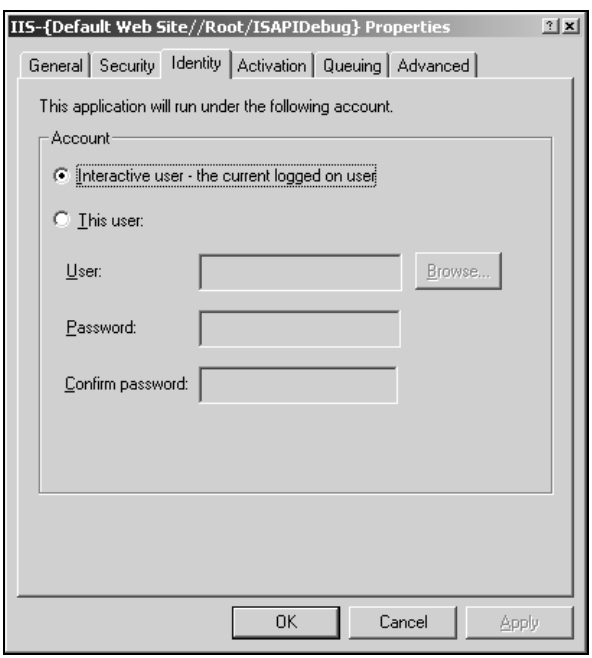


а

Рис. 16.9. Этапы настройки ISAPI-приложения для отладки в Delphi:
а — оснастка настройки приложений COM+

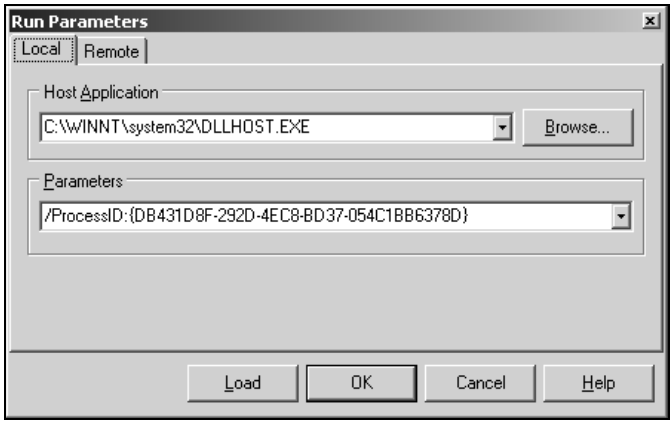


б



в

Рис. 16.9. Этапы настройки ISAPI-приложения для отладки в Delphi:
б — диалоговое окно с указанием `Application ID` нужного экземпляра dllhost.exe;
в — настройки идентификации



г

Рис. 16.9. Этапы настройки ISAPI-приложения для отладки в Delphi: *г* — настройки параметров запуска приложения в среде Delphi

Резюме

Технология Web Broker позволяет создавать серверные Web-приложения нескольких основных типов для популярных серверов Microsoft Internet Information Server и Apache. Набор специализированных компонентов и модулей обеспечивает приложению выполнение следующих функций:

- взаимодействие с Web-сервером, обмен запросами и ответами;
- управление выделением памяти и диспетчеризацию 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.

Глава 17

Протокол SOAP и Web-службы. Клиентская часть

Увидев в Delphi целые страницы с новыми компонентами для Internet, папки с десятками файлов исходных текстов и сотни страниц документации, каждый пользователь вправе задаться вопросом: а зачем это и почему нельзя обойтись существующими технологиями? Вопрос справедливый, ибо избыточность инструментальных средств — верный путь к их неправильному применению.

И, тем не менее, появление этих новых технологий — не просто дань моде, а очень и очень насущная необходимость. Обусловлена она хорошо известной проблемой — обеспечением совместимости приложений, работающих на разных платформах. Кажется, в последние два года средство для вызовов методов удаленных приложений найдено и, главное, одобрено большинством IT-сообщества. Называется оно Simple Object Access Protocol (SOAP). О нем и пойдет речь в этой главе. Сначала мы убедимся в необходимости протокола SOAP и найдем его место среди других технологий. Потом рассмотрим основы SOAP и Web-служб — приложений на его основе. Шаг за шагом пройдем по пути создания клиентского и серверного приложения SOAP в Delphi. Вопросам же взаимодействия Web-служб, созданных в Delphi, с программными решениями других поставщиков, будет посвящена целиком следующая глава.

17.1. Почему SOAP?

Фирмы, занимающиеся программным обеспечением, и независимые разработчики отнюдь не вчера начали трудиться над созданием языка, на котором могли бы разговаривать в Internet Windows и UNIX, Macintosh и мэйнфреймы. Конечно, прочитав предыдущие главы, посвященные сетевым протоколам, вы назовете HTTP и ряд других высокоуровневых протоколов. Но что делать, если нужно забраться еще выше? Если необходимо не просто запросить гипертекст и показать его пользователю, а организовать взаимо-

действие двух одновременно работающих на разных платформах приложений?

Существуют решения и для этой задачи — например, CORBA. Но у этой действительно кросс-платформенной 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 (ORB, Object Request Broker) — вам следует отдельно позаботиться о приобретении прав на него.

Ярким примером другого подхода можно считать созданное несколько лет назад решение фирмы 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 тут — ключевой момент предложенного решения. Уровень безопасности в Internet становится все более и более строгим, 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 сетевыми атаками 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 наивно предполагать, что так же поступят все ваши потенциальные клиенты, если их у вас хотя бы десятки или сотни. Следовательно, единственным действительно общепринятым на сегодня протоколом для установления соединения по Internet следует признать HTTP.

CIS позволяет обойти брандмауэры, несмотря на строгие меры безопасности. Но это решение ориентировано только на DCOM — о совместимости с другими платформами говорить не приходится.

17.1.1. Delphi и *TSomeConnection* — потенциальные проблемы

Давайте посмотрим с высоты этих глобальных проблем на компоненты Delphi для работы с удаленными базами данных (технология DataSnap, ранее называвшаяся MIDAS; *см. гл. 8*). Архитектура DataSnap такова: на серверной стороне есть объект, реализующий интерфейс `IAppServer`,

на клиентской — объект `TClientDataSet`, содержащий ссылку на `IAppServer`. В качестве последнего выступают компоненты соединения — `TDCOMConnection`, `TSocketConnection`, `TWebConnection` и т. д. Такие компоненты также содержат реализацию `IAppServer`; но это не "настоящий" сервер, а средство для пересылки (маршаллинга) запросов от клиента к удаленному приложению. Иными словами, клиент просто обращается к `IAppServer`, не заботясь о том, как данные будут упакованы, переданы, проверены и доставлены.

На серверной стороне маршаллинг реализован по-разному. Объект `TDCOMConnection` не требует серверной части — вызов удаленных интерфейсов производит сама система DCOM. Но поддержка CIS, в данном случае фирмой Borland, не предусмотрена, поэтому исключаем `TDCOMConnection` из числа кандидатов для работы 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 (принят 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 предоставить 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.

17.2. SOAP : краткое содержательное описание

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 метафора. 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 и не зависел 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 стандартизировать соответствующую привязку (binding). Но трудно представить себе выражение `y:=sin(x)`, вычисляемое по почтовому запросу — это бесполезно, поскольку слишком медленно. Изначальная спецификация SOAP документирует только привязку к HTTP. Так эти два протокола в сознании многих и стали неразрывными.

Приведем пример посылки SOAP:

```
<?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 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.

```
<?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, то есть правила написания писем, — это первая часть стандарта.

17.2.1. Web-службы

Теперь мы можем дать определение Web-службы. Это — серверное приложение, доступное через Internet, чей интерфейс опубликован в соответствии со спецификацией 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 функции: получение данных 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).

В отличие от COM-объектов, о том, 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.

17.2.2. Описание протокола WSDL

Протокол WSDL (напоминаем, расшифровывается он как Web Services Description Language) — и есть тот язык, на котором для клиента публикуется описание Web-службы. 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 нижеследующих базовых понятий.

1. *Перечень типов (Types).* Основой для обмена данными являются типы, стандартизованные для XML. Но если ваша служба применяет нестандартные (custom) типы — например, структуры данных — их следует описать, указав или приложив соответствующую XML-схему (формата XSD).
2. *Сообщение (Message).* Каждое сообщение — это "атом", которыми обмениваются соединяющиеся стороны. Сообщение состоит из одной или более частей (параметров). Например, вы вызываете на серверной стороне функцию, чье описание на Object Pascal 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`.
3. *Тип порта (Port type).* Тип порта — это описание некоторого набора операций. Операция — это действие над входным сообщением, порождающее выходное сообщение; очень похоже на механизм RPC (Remote Procedure Call). Для полноты картины упомянем, что стандарт WSDL определяет и обратный порядок вызова: сервер посылает сообщение, клиент отвечает.
4. *Привязка (Binding).* Привязка задает связь между типом порта и транспортным протоколом, с помощью которого будут доставляться сообщения.
5. *Порт (Port).* 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.
6. *Служба (Services).* Наконец, собственно служба — это поименованный набор портов.

Взаимосвязь этих базовых понятий показана на рис. 17.1.

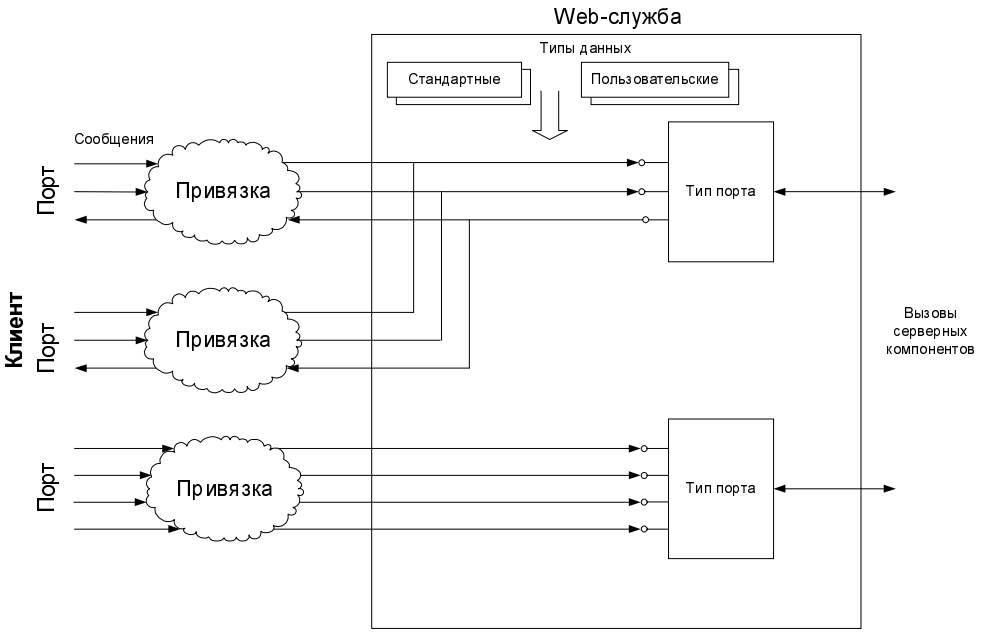


Рис. 17.1. Базовые элементы Web-службы в терминах WSDL

Документы WSDL, что естественно, описываются на XML. Все упомянутые 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 Web-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 какой-либо службы, поищите такие элементы в ее WSDL-документе.

Давайте разберем пример WSDL-файла для службы, которая способна делать две простые вещи — возвращать в неизменном виде строку и массив строк, присланные ей для обработки (листинг 17.1). В дальнейшем мы будем создавать Web-службу на основе этой спецификации.

Листинг 17.1. 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" transport="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 location=
"http://localhost/scripts/simpleecho.dll/soap/ISimpleEcho"/>
    </port>
  </service>
</definitions>
```

Из чего состоит этот файл?

1. 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`.
2. Раздел сообщений содержит четыре элемента — по одному для входного и выходного параметров двух наших функций. Имена их, соответственно, `echoStringRequest`, `echoStringResponse`, `echoStringArrayRequest`, `echoStringArrayResponse`. Теоретически возможно, чтобы одно и то же сообщение использовалось в описании нескольких типов портов; однако на практике стараются этого избегать.
3. Единственный тип порта называется `ISimpleEcho` (обратите внимание на это, позже мы будем объяснять, как связаны параметры проекта в Delphi с параметрами службы). Он содержит описания двух операций — `echoString` и `echoStringArray`.
4. Привязка указывает на то, что для доставки сообщений будет использоваться протокол SOAP.
5. 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 открытого 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 в 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. 17.2).

Если вам нужен только WSDL-файл в формате XML без всякого красивого обрамления, нужно задать путь вида **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;
```

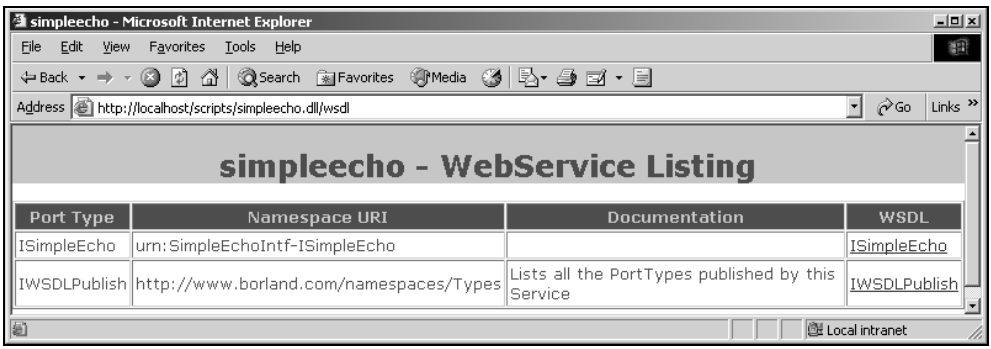


Рис. 17.2. Информация о собственных портах, представленная Web-службой, созданной в Delphi

Зная, что серверная часть службы написана средствами Delphi, вы можете обратиться к ее порту `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 сообщениями заданных типов? Ответ на этот вопрос зависит от применяемой вами среды разработки. К счастью, в случае Delphi он очень прост. К рассмотрению архитектуры клиентской части мы сейчас и переходим.

17.3. Архитектура Web-служб в Delphi

Как и многие другие технологии, SOAP для разработчика в Delphi может предстать в разных обличьях. Если вы хотите создать клиента для удаленных баз данных, вы можете просто взять компонент `TSOAPConnection` и использовать SOAP как один из возможных протоколов уровня соединения, наряду с HTTP, CORBA, DCOM или сокетами. В этом случае специальные знания вам почти не понадобятся — вся функциональность протокола SOAP инкапсулирована внутри упомянутого 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, вы можете использовать целый ряд компонентов "нижнего" уровня, которые напрямую формируют и принимают посылки протокола SOAP.

Наконец, как клиент, так и сервер могут быть написаны с использованием других инструментов. Следующая глава будет посвящена взаимодействию Delphi со средствами для создания Web-служб фирмы Microsoft.

Компоненты, которыми вас снабдили разработчики фирмы Borland для работы с Web-службами, находятся на одноименной странице. Помимо этого, создавая новый проект Web-службы, начните с вызова пункта меню **File | New | Others**. В Репозитарии объектов, на странице **Web-Services**, 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.

При установке Delphi вы видели сообщение, предупреждающее о том, что SOAP и Web-службы являются развивающимися технологиями, и стандарты в этой области могут изменяться. Что реально стоит за этим предупреждением? Важное практическое замечание: начальная версия Delphi 6, да и update 1, содержали слишком "сырую" реализацию SOAP, которая к тому же имела проблемы взаимодействия со службами, созданными на основе Microsoft SOAP Toolkit. Только начиная с Update pack 2, эта новая технология получила в Delphi более или менее надежную основу.

Примечание

Однако ошибки и несоответствия в таких развивающихся технологиях все же дело обычное. Уже после выхода Update Pack 2 в реализации SOAP произошли изменения. Они, например, исправляют ошибку несовместимости Borland SOAP с установленным анализатором MSXML4. К сожалению, если вы или ваши клиенты используете этот анализатор, ручные исправления абсолютно необходимы — *см. статью по адресу*: **http://community.borland.com/article/0,1410,28514,00.html**. Впрочем, когда вы читаете эти строки, уже может быть доступна следующая версия или Update Pack. В любом случае, для надежной работы вам следует почаще заходить на сайт фирмы Borland за свежими обновлениями.

17.4. Клиент Web-службы

Изучение 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 знакомство с ней поможет нам плавно подойти к написанию серверной части.

Итак, мы начинаем рассмотрение клиента, еще не создав собственного сервера. Однако мы полагаем, что с доступом в 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/**.

Назначение 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.

17.4.1. Генерация интерфейса Web-службы

Первое, с чего следует начинать создание клиента службы — вызов мастера импорта WSDL-файлов из Репозитория Объектов (меню **File | New | Other**, страница **WebServices**). Подразумевается, что у вас уже есть готовый открытый проект — это может быть 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 ранее (рис. 17.3).

Файл описания может находиться как в Сети, так и локально на вашем компьютере.

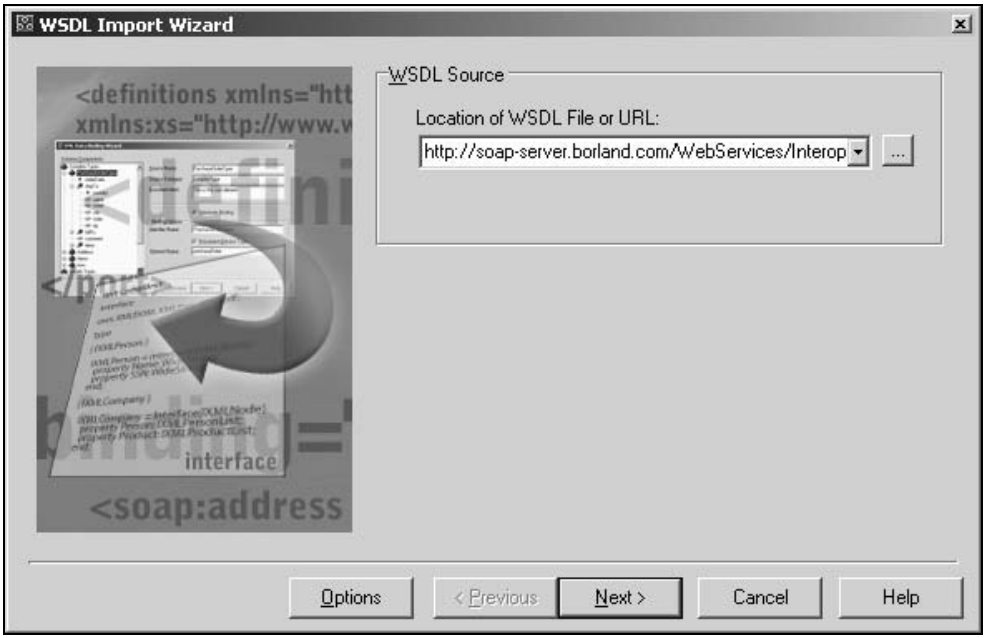


Рис. 17.3. Мастер импорта WSDL-файлов — страница ввода адреса службы

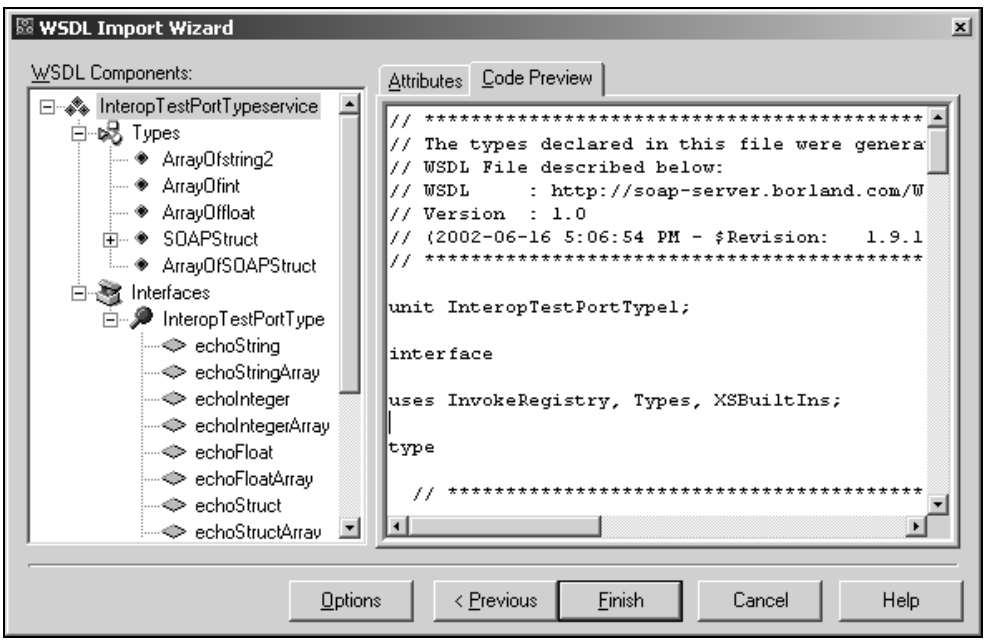


Рис. 17.4. Мастер импорта WSDL-файлов — страница интерфейса службы

На втором шаге Мастер уже готов показать вам результаты своей работы (рис. 17.4). 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 Object Pascal, описывающий службу. Нажмите кнопку **Finish** — и сгенерированный Мастером файл добавится в ваш текущий проект.

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 найти и отредактировать.

Сохраним файл и проанализируем его содержимое. После описания структур данных следует собственно интересующий нас интерфейс:

```
  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): Ar-
rayOfint; stdcall;
    function  echoFloat(const inputFloat: Single): Single; stdcall;
    function  echoFloatArray(const inputFloatArray: ArrayOffloat): Array-
Offloat; stdcall;
    function  echoStruct(const inputStruct: SOAPStruct): SOAPStruct; stdcall;
    function  echoStructArray(const inputStructArray: ArrayOfSOAPStruct):
ArrayOfSOAPStruct; stdcall;
    procedure echoVoid; stdcall;
    function  echoBase64(const inputBase64: TByteDynArray): TByteDynAr-
ray; stdcall;
    function  echoDate(const inputDate: TXSDateTime): TXSDateTime; stdcall;
    function  echoHexBinary(const inputHexBinary: TXSHexBinary): TXSHex-
Binary; 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 SOAPBuilders Interoperability Lab для тестирования. Некоторые из них (`Integer`, `Boolean`, `string`) совпадают с оригинальными (native) типами данных Object Pascal.

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`: вы наверняка будете использовать `TXSDateTime`, `TXSDecimal`, `TXSHexBinary` и т. п. Задача 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. О типах данных мы будем говорить позже.

Специально подчеркнем связь мастера импорта WSDL-файлов с технологией COM. При всем внешнем сходстве приведенный интерфейс НЕ соответствует реальному COM-объекту. Ни вызов `CreateOLEObject` с параметром `'InteropTestPortType'`, ни вызов `CreateCOMObject` с `GUID {426150D1-7C2C-A74F-954B-DB3ADC067CD8}` ничего вам не возвратят. Соответственно, содержимое системного реестра не изменяется: в нем эти интерфейсы не регистрируются. Вы можете импортировать описания десятков служб; единственное, что потом придется удалить, это созданные PAS-файлы.

Тем не менее, технология COM здесь все же присутствует. В исходном коде 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, позже заглянем в "нутро" этого объекта, который служит "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 объект `InteropTestPortType`. 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 методом `GetInteropTestPortType` — получите первый привет из мира Web-служб.

В примере мы вызвали функцию `GetInteropTestPortType` без параметров, а между тем их у нее два, хоть и необязательных. Они нужны для того, чтобы создавать объект, применяя не тот WSDL-файл или URL, который использовался Мастером импорта. Допустим, у вас на диске хранится копия WSDL-документа, предназначенная для отладочных целей (такая возможность рассмотрена далее). В этом случае параметр `UseWSDL` должен быть равен `True`, а параметр `Addr` — указывать на отладочный файл:

```
Port := GetInteropTestPortType(True, 'c:\MyFolder\InteropTestPortType.wsdl');
```

Итак, первый шаг в мир Web-служб можно считать сделанным. Ваших знаний теперь достаточно для того, чтобы найти описание интересующей службы, создать для нее клиента и вызвать методы. Но вряд ли вас удовлетворят только самые простые случаи — поэтому идем дальше.

17.4.2. 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 можем знать 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 формировать динамически, а не статически? Компилятор знает структуру ТВМ, потому что он обработал PAS-файл описания интерфейсов службы, сгенерированный вышеупомянутым Мастером. Откуда в ней появятся методы и как они будут вызываться, ему все равно.

В фирме 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 на Object Pascal

```
Obj := RIO as InteropTestPortType;
```

то RIO попытается сделать следующее.

1. Считает описание затребованного интерфейса из специального хранилища `InvokeRegistry`. В нем хранятся описания всех интерфейсов Web-служб; подробнее о нем речь пойдет чуть позже.
2. По полученному описанию сгенерирует ТВМ и методы-переходники (Stubs), которые предназначены для приведения в соответствие правил соглашения о вызове (calling convention), а также типов параметров данного метода.
3. Вернет ссылку на созданную ТВМ и увеличит СВОЙ счетчик обращений на единицу — отныне RIO работает как объект, поддерживающий запрошенный интерфейс.

Сам класс `TRIO` — не более чем общая схема реализации идеи удаленного вызова. В Палитре компонентов Delphi вы найдете компонент `THTTPRIO`, который адаптирует `RIO` для передачи 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.

Создать этот необычный объект можно двумя способами.

1. Как обычный компонент. Поместите на форму экземпляр `THTTPRIO` из Палитры и задайте его свойства в режиме разработки. В этом случае хозяином (Owner) компонента будет форма, фрейм или другой компонент. Он же будет отвечать за освобождение экземпляра `THTTPRIO`.
2. Динамически. Вызывается 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 объекта и сообщаете библиотеке времени выполнения (RTL) Delphi о необходимости вызвать метод `_Release` и уменьшить число обращений. Когда счетчик становится равным нулю, объект должен уничтожиться.

Но код компонента `TRIO` и его потомки написаны таким образом, что объект освобождается 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 для чтения:

```
property RefCount: Integer;
```

Если вы не уверены в своем коде, контролируйте значение `RefCount`: перед вызовом `TRIO.Free` значение числа ссылок должно быть нулевым.

17.4.3. 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 раскрывающийся список покажет вам адреса уже знакомых служб. Напомним, что адрес может задаваться не только URL, но и полным именем локального файла 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 в Delphi или Kylix. Если 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 лишь разницей, что действие `/wsdl` заменяется на действие `/soap` (как в только что приведенном примере).

Естественно, свойства URL и `WSDLLocation/Service/Port` являются взаимоисключающими — при настройке одного значение другого (других) очищается.

Проиллюстрируем применение этих свойств 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 в ее обработку события `OnClick` следующий код:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
   ShowMessage((HTTPRIO1 as InteropTestPortType).echoString('Hello,
world!'));
end;
```

При таком использовании заботиться о счетчике обращений не обязательно — код очистки RTL Delphi, выполняемый при выходе из метода `Button2Click`, приведет его в порядок.

У объекта `THTTPRIO` есть всего два события — `OnBeforeExecute` и `OnAfterExecute`:

```
TBeforeExecuteEvent = procedure(const MethodName: string;
var SOAPRequest: InvString) of object;
property OnBeforeExecute: TBeforeExecuteEvent;
TAfterExecuteEvent  = procedure(const MethodName: string; SOAPResponse:
TStream) of object
property OnAfterExecute: TAfterExecuteEvent;
```

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 не что иное, как посылки одноименного протокола.

17.4.4. Решение коммуникационных проблем

Надежность и скорость работы служб зависят прежде всего от транспортного протокола, доставляющего сообщения. В данном случае это HTTP. В Delphi 6 HTTP реализован с помощью вызова библиотеки WinInet.dll, поставляемой вместе с Windows и Internet Explorer и содержащей весь механизм для работы с высокоуровневыми протоколами, встроенными в IE — HTTP, HTTPS, FTP, 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 говорили в предыдущих главах). Если вы посмотрите на код модуля SoapHTTPTrans.PAS (папка Source\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, краткий обзор свойств компонента `THTTPRIO` оказался практически исчерпывающим. Зато у него есть два дочерних компонента со своими (важными!) свойствами — в Инспекторе Объектов вы видите их как `Converter` и `HTTPWebNode`.

Если вы хотите познакомиться со всеми шестью компонентами на странице **WebServices** Палитры компонентов Delphi, то вот вам еще два: `TOPToSoapDomConvert` и `THTTPReqResp`. Они могут использоваться самостоятельно, и в то же время они входят в состав `RIO` — это, как вы наверное догадались, и есть `Converter` и `HTTPWebNode`.

Первый компонент — `Converter` — отвечает за преобразование вызовов между протоколом SOAP и приложением. Самое важное опубликованное свойство компонента — `Options`. Его настройки в основном влияют на совместимость с другими клиентами и серверами Web-служб. Поскольку в Delphi они общие и для компонента `Converter`, и для серверов служб, мы рассмотрим их подробнее в следующей главе.

Второй компонент — `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 того сервера, через который будет происходить соединение. Внимание: даже если свойство не задано, это НЕ ОЗНАЧАЕТ, что соединение будет происходить без прокси-сервера. 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 прокси-сервер, 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 о том, чтобы в этом свойстве было указано имя агента 'Borland SOAP 1.1' (и 'DataSnap', если вы пользуетесь компонентом `TWebConnection`).

2. Авторизация пользователя службы. Если сервер службы или прокси-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 (например, если приложение запускается автоматически); задав значения этих свойств, вы можете избежать ввода имени и пароля вручную.

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 сервером кода ошибки 401 и генерации соответствующей исключительной ситуации. Динамически созданный `THTTPRIO`, очевидно, не инициализирует значений этих свойств: вы увидите системный диалог авторизации, внешний вид которого зависит 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: библиотека WinInet должна правильно распознать настройку.

Примечание

Если вы же пишете серверную часть службы, синхронно с аналогичным изменением в приложении измените название протокола в WSDL-файле — не сделавшие этого совершают типичную ошибку.

В свойстве `InvokeOptions` компонента `THTTPReqResp` 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 информирование пользователя.

Если вы вдобавок используете для авторизации клиентские сертификаты, то ситуация усложняется. Класс THTTPReqResp не содержит способов выбора нужного сертификата из нескольких возможных. К тому же, разные реализации WinInet ведут себя с клиентскими сертификатами по-разному: одни не поддерживают эту возможность или содержат ошибки, вторые автоматически выбирают первый доступный из клиентских сертификатов, третьи открывают для этого диалоговое окно выбора. Все это следует иметь в виду; если вы все же решились защитить службу с помощью такого метода авторизации, очевидно, вам придется частично переписать код модуля SOAPHTTPTrans. Функция из WinInet под названием `InternetSetOption` позволяет выбрать контекст нужного сертификата.

Примечание

То же самое можно сказать и еще об 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, так как они не являются виртуальными и перекрыть их нельзя, и кроме того, первый раз они вызываются еще до первого вызова методов службы.

17.4.5. Что такое *Invoke Registry*. Регистрация интерфейсов и типов данных

Если вы внимательно просмотрели 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)? Приведем фрагмент исходного текста 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 реализация навигации по RTTI для конкретного экземпляра класса.

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 проекта 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.

Многие в первый раз слышат о директиве {$M}, потому что она работает таким образом: если родительский класс был скомпилирован в состоянии {$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? То, что данный интерфейс и все классы Delphi, реализующие его, будут включать информацию о методах интерфейса в RTTI, как будто они — опубликованные методы самого класса.

Такое новшество появилось впервые в Delphi 6 и разработано оно специально для создания Web-служб и их клиентов. В самом деле, сходство задач очевидно: 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 одного механизма 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, содержащие описание интерфейса при помощи Мастера, то регистрация будет выполнена без вашего участия — посмотрите на секцию инициализации сгенерированного для примера модуля InteropTestPortType1.pas.

Второй объект — `RemClassRegistry` — хранит описания зарегистрированных типов данных. Неверно 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 автоматически. В нашем примере со службой BASE регистрации подлежат структура SOAPStruct и массив таких структур.

И тут нас подстерегает 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 способом, известным библиотеке времени выполнения RTL Delphi.

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 он 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 6 снабдили нас такими классами — они находятся в модуле XSBuiltIns и называются, соответственно, TXSTime, TXSDate TXSDateTime, 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 обращаться к этим методам. 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. В этом случае работа с ними почти не будет отличаться от уже описанного способа использования SOAPStruct:

- создайте экземпляр класса;
- при помощи AsSomething присвойте ему нужное значение;
- передайте его в качестве параметра, за преобразование он отвечает сам;
- если служба возвращает значение этого типа, прочитайте полученный результат при помощи того же метода.

Предположим, что ваша задача не может быть решена при помощи Мастера и описанных выше типов, тогда класс, представляющий ваш тип данных, придется писать самому. Первый совет, который нужно дать здесь, — ...не делайте этого. Мы уже не раз говорили о проблеме совместимости реализаций SOAP. Не факт, что другие средства разработки смогут правильно представить и обработать ваш тип данных. По возможности, следует ограничиться стандартными типами.

Если же это невозможно, надо 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 "поглощать" 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 18

Серверная часть Web-службы. Совместимость

В этой главе мы продолжим разговор о создании Web-службы. Разработка сервера является, как правило, более объемной и сложной задачей, так как именно он отвечает за реализацию всей функциональности — протокол SOAP лишь доставляет пользователю результаты.

Следующий важнейший аспект, обсуждающийся в этой главе, — совместимость. Мы рассмотрим здесь Microsoft SOAP Toolkit — инструмент для публикации объектов Автоматизации в качестве Web-служб. Создав вторую пару из клиента и сервера с аналогичными возможностями, мы сможем произвести их "перекрестное" соединение и воочию убедиться в том, что Web-службы придуманы не в Borland и что предлагаемые вам в Delphi решения действительно умеют "общаться" с другими платформами.

И наконец, эта глава познакомит вас с удобным (и практически единственным) средством для отладки служб — утилитой SOAP Trace.

18.1. Создание тестового примера — службы *SimpleEcho*

Из серверных заготовок в Репозитории объектов основной является **SOAP Server Application**. Две прочие — **SOAP Server Data Module** и **SOAP Server Interface** — составные части основной.

Итак, первый вопрос, на который предстоит ответить, выбрав **SOAP Server Application**, — тип серверного приложения. Выбор из пяти вариантов на первый взгляд сложен. Но на самом деле все просто — давайте лишь разберемся. Если вашим Web-сервером является Apache, то ваш выбор — Apache Shared Module. Если вы работаете с Microsoft IIS, первый вариант — ISAPI DLL — предпочтительный вид приложения для него. Приложения CGI и WinCGI — это исполняемые модули, которые запускаются в отдельных процессах, что сильно повышает системные накладные расходы; их используют,

как правило, в тех случаях, когда остальные варианты по тем или иным причинам недоступны или нежелательны. Наконец, если вы хотите отлаживать серверную часть, фирма Borland включила в состав Delphi специальную утилиту — WebApp Debugger.

Мастер создает главный Web-модуль приложения (модуль данных 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 интересный и полезный пример. Если вы доросли до реального приложения, к этому моменту вы уже будете знать, какие интерфейсы и как вам надо опубликовать. О вариантах публикации — позже, а пока продолжим создание нашей первой Web-службы.

Итак, диалог создания новой Web-службы выглядит, как показано на рис. 18.1. Имя службы — идентификатор, под которым она будет известна клиентам. Логично задать для создаваемых модулей Delphi тот же идентификатор. Если точнее, то модулей будет создано два — в данном случае `SimpleEchoInft.pas` и `SimpleEchoImpl.pas`.

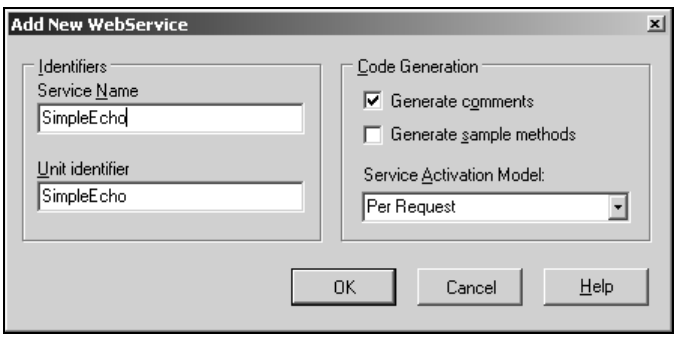


Рис. 18.1. Диалоговое окно параметров интерфейса Web-службы

Почему два? Суффиксы `Intf` (Interface) и `Impl` (Implementation) приводят нас к ответу. Интерфейсная часть службы понадобится как серверу, так и клиенту (если, разумеется, этот клиент написан в 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 в Delphi, достаточно WSDL-файла — иначе как бы такие клиенты соединялись со службами других поставщиков? Но мастер импорта WSDL *(см. описание клиентской части в предыдущей главе)* создаст для вас точно такой же файл интерфейса, какой уже есть на сервере (в данном примере — аналог `SampleServiceInft.pas`). Есть только одно отличие — функция, возвращающая экземпляр RIO на клиентской стороне.

Флажок **Generate sample methods** включит в сгенерированный интерфейс примера четыре метода:

```
ISimpleEcho = 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.

В создаваемой нами в качестве примера службе SimpleEcho мы ограничимся всего двумя подобными методами — `echoString` (возвращает присланную строку) и `echoStringArray` (то же для массива строк). В конце концов, эта глава посвящена вовсе не науке вычислений или, скажем, операциям с базами данных. Как научить приложение выполнять такие операции, вы, вероятно, уже знаете или узнали, прочитав предыдущие главы. Здесь мы учимся обмениваться уже готовыми результатами посредством SOAP. Поэтому нет ничего страшного в том, что код методов службы предельно прост:

```
function TSimpleEcho.echoString(const Value: String): String;
 begin
  Result := Value;
 end;


 function TSimpleEcho.echoStringArray(const Value: TStringDynArray):
TStringDynArray;
 begin
  Result := Value;
 end;
```

Итак, после добавления интерфейса перед нами Web-модуль с тремя расположенными на нем компонентами (рис. 18.2).

Сервер в его простейшем виде "готов к употреблению". Скомпилируем нашу DLL (в случае, если вы выбрали тип приложения ISAPI) и поместим ее в одну из папок Web-сервера. Надо только помнить, что у нее должны быть

установлены разрешения на запуск сценариев и исполняемых файлов. Если вы размещаете приложение в папке scripts, которая автоматически создается IIS и предназначена как раз для этих целей, то проблем нет. Если же вы решили отвести под исполняемые модули службы отдельную папку, не забудьте настроить права доступа.

Рис. 18.2. Web-модуль, создаваемый мастером SOAP Server Application

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 пакет обновления (hot fix), описанный в **http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/ms02-018.asp**. Без этого работу начинать опасно.

18.2. 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. 18.2).

Компонент `TWebModule`, как известно, играет роль централизованного диспетчера и обработчика 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 — это `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. Оно 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`.

18.2.1. 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, вы 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. 18.3).

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, просто набрав **http://localhost/scripts/simpleecho.dll**.

Из каких колонок состоит таблица на рисунке? 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. 18.3).

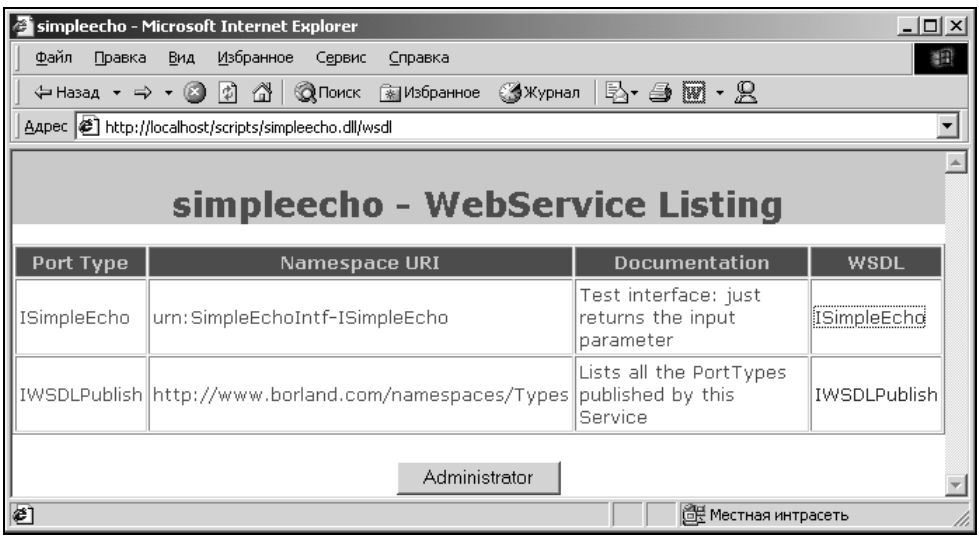


Рис. 18.3. HTML-документ с описанием портов службы, генерируемый компонентом `TWSDLHTMLPublish`

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 установлено в `True`, под списком интерфейсов появляется кнопка **Administrator**, которая видна на рис. 18.3.

Нажав кнопку, вы попадаете в простой редактор портов (рис. 18.4). Собственно, больше ничего на этом этапе поменять нельзя: 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/изменить.

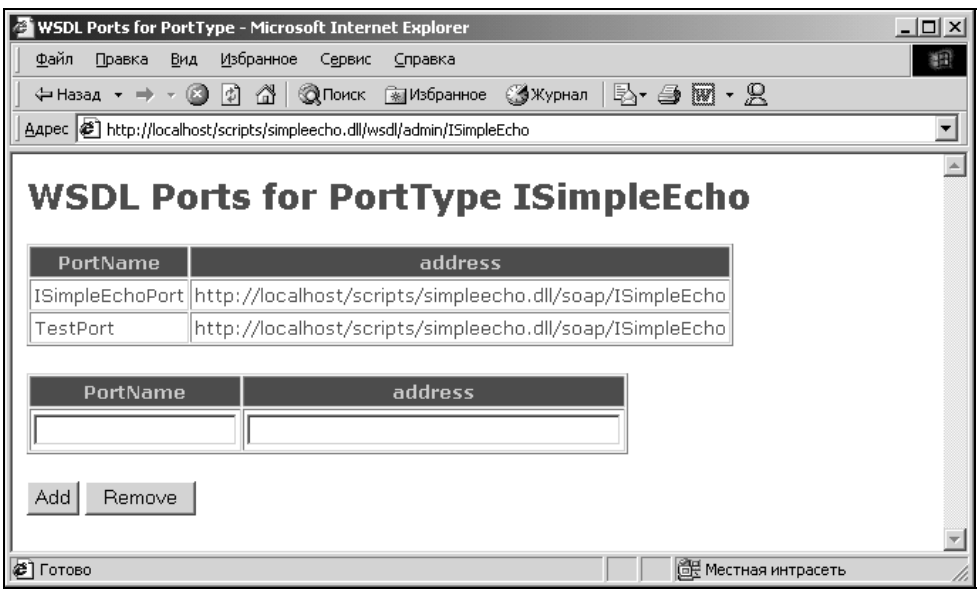


Рис. 18.4. Администрирование списка портов в браузере

В примере на рисунке 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.

18.2.2. 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 сколь-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:

```
TBeforeDispatchEvent = procedure(const MethodName: string; const Request:
TStream) of object;
property BeforeDispatchEvent: TBeforeDispatchEvent;
TAfterDispatchEvent = procedure(const MethodName: string; SOAPResponse:
TStream) of object;
property AfterDispatchEvent: TAfterDispatchEvent;
TOnExceptionEvent = procedure(const MethodName: string; const Request:
TStream; const Response: TStream) of object;
property OnExceptionEvent: TOnExceptionEvent;
```

Во-вторых, этот компонент имеет набор параметров (свойство `Options`), определяющий совместимость.

18.2.3. 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. 18.5; код приводится на прилагаемом к книге носителе).

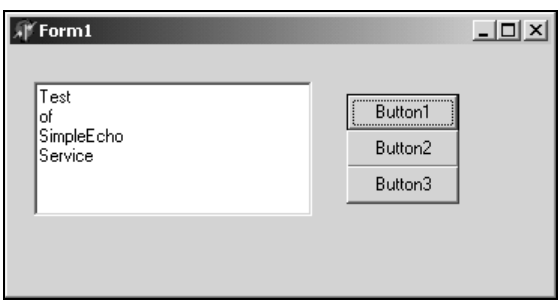


Рис. 18.5. Главное окно клиента службы `SimpleEcho`

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.

18.2.4. 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 определились с тем, что для этого интерфейсы должны 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)*, то есть отображения пакетов SOAP на вызовы функций и обратно. Этой ИС можно ждать при любых отклонениях от стандарта SOAP, что, увы, не редкость.

18.2.5. Несколько слов об использовании *TSOAPConnection*

Нерассмотренным остался последний, седьмой компонент со страницы **WebServices** Палитры инструментов — это `TSOAPConnection`. Он принадлежит к следующему уровню иерархии. Если предыдущие компоненты обеспечивали соединение посредством SOAP, то этот использует готовую реализацию SOAP для передачи пакетов DataSnap.

Инженерами Borland были подтверждены (**http://groups.google.com/groups?q=TSOAPConnection+group:borland.public.delphi.webservices.soap&hl=ru&lr=&ie=UTF-8&inlang=ru&selm=3c6dabdb%241_1%40dnews&rnum=1**) следующие принципиальные недостатки текущей реализации `TSOAPConnection`.

1. Вызываемым (Invokable) интерфейсом является `IAppServer`. Если вы породили от него дочерний (child) интерфейс с добавлением собственных методов, то они не могут быть вызваны, так как `TSOAPConnection` всегда запрашивает создание `IAppServer`, а не его потомков.
2. Если серверное приложение содержит более одного модуля данных (соответственно, более одного класса, поддерживающего `IAppServer`), то нет гарантии, что `TSOAPConnection` произведет соединение с правильным модулем данных; по причине, изложенной в пункте 1, они для него неотличимы.

Все изложенное приводит к очевидному выводу: по состоянию на сегодняшний день `TSOAPConnection` еще не может служить решением для реальных приложений. Тем более что есть компонент `TWebConnection`, который сочетает преимущества работы с применением протокола HTTP

со скоростью и надежностью. Однако можно быть уверенным, что перечисленные недостатки будут устранены в ближайшем пакете обновлений (update pack) и SOAP совместно с DataSnap еще помогут разработчикам распределенных приложений.

18.3. Средства разработки для SOAP: подход Microsoft

Было бы странно писать книгу о новейших решениях для обеспечения кросс-платформенной совместимости между приложениями и рассмотреть только одно 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. Наконец, совсем недавно появилась бета-версия 3.0, со многими исправлениями и функциональными расширениями.

Нельзя не упомянуть о .NET — принципиально новой архитектурной концепции Microsoft. Она во многом основана на идее Web-служб. Однако материал о ней и средстве разработки Visual Studio.NET мог бы сделать эту главу и всю книгу во много раз толще. Прототипы новых средств разработки Borland для .NET к моменту написания книги уже были показаны общественности, поэтому разумно не включать информацию о .NET в книгу о Delphi до появления полноценных продуктов.

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 использован 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 ту, которая применена 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.

18.3.1. Краткий обзор архитектуры Web-служб на 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, экспонируемый в качестве службы, должен поддерживать интерфейсы — потомки `IInvokable`. В SOAP Toolkit требование таково: объект должен поддерживать `IDispatch`, то есть 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 входит 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 играет ту же роль, 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' binding='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 прямо в этом же модуле. SOAP Toolkit использует один общий модуль расширения — слушатель. Слушатель вызывает методы разнообразных COM-объектов, которые несут в себе функциональность 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`.

18.3.2. 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 гл. 17)*. 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 18.1 (впрочем, этот файл есть среди примеров на носителе, поставляемом с книгой).

Листинг 18.1. 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, так как 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).

18.3.3. 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":

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 **New Automation Object**. Назовите его `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. 18.6.

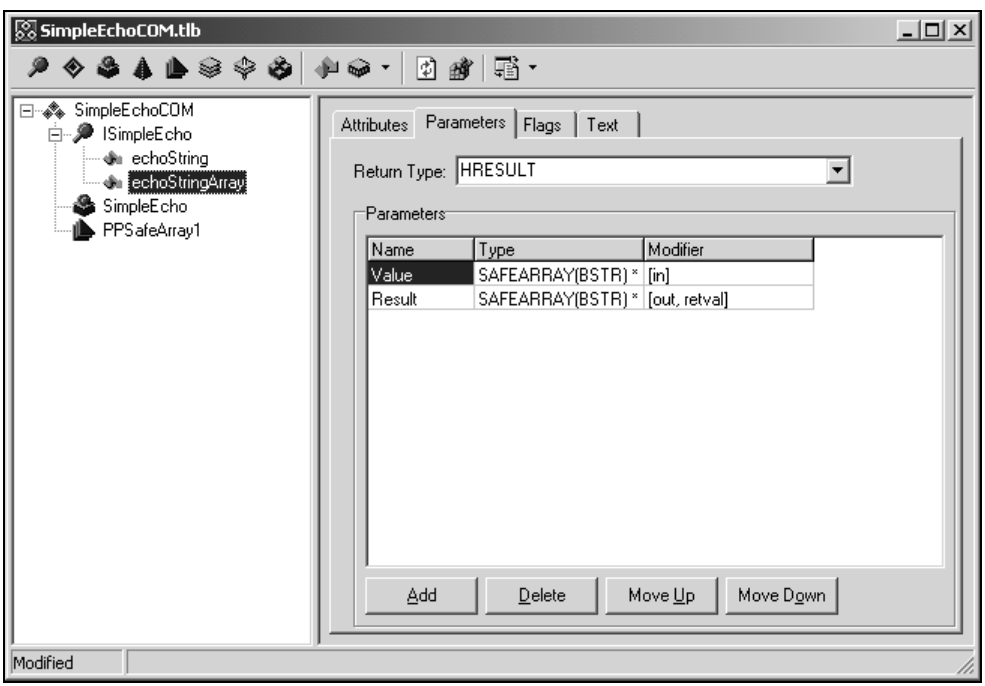


Рис. 18.6. Внешний вид библиотеки типов проекта SimpleEchoCOM

Массивы строк, передаваемые в метод `echoStringArray`, должны иметь тип `SAFEARRAY(BSTR)*` (мы не можем применить класс `TStringDynArray`,

так как 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, полный исходный код которого прилагается к книге, обеспечивает 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. гл. 17)*, 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.

18.4. 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; она 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"* 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.

Диалоговое окно настройки параметров перехвата выглядит следующим образом (рис. 18.7).

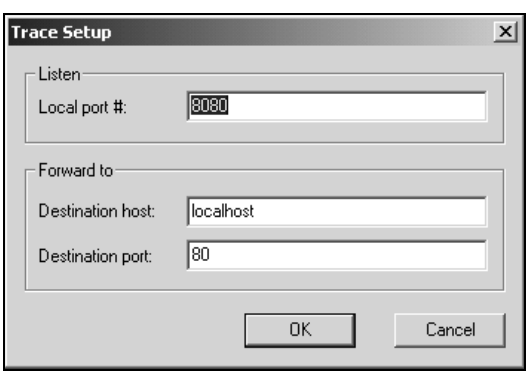


Рис. 18.7. Настройка параметров утилиты SOAP Trace

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. 18.7, вообще менять не надо: пакеты будут переадресовываться с адреса **localhost:8080** на **localhost:80**, а это именно то, что требуется.

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 выше примере **http://soap.myservices.com/scripts/** следует поменять на **http://localhost:8080/scripts/**.

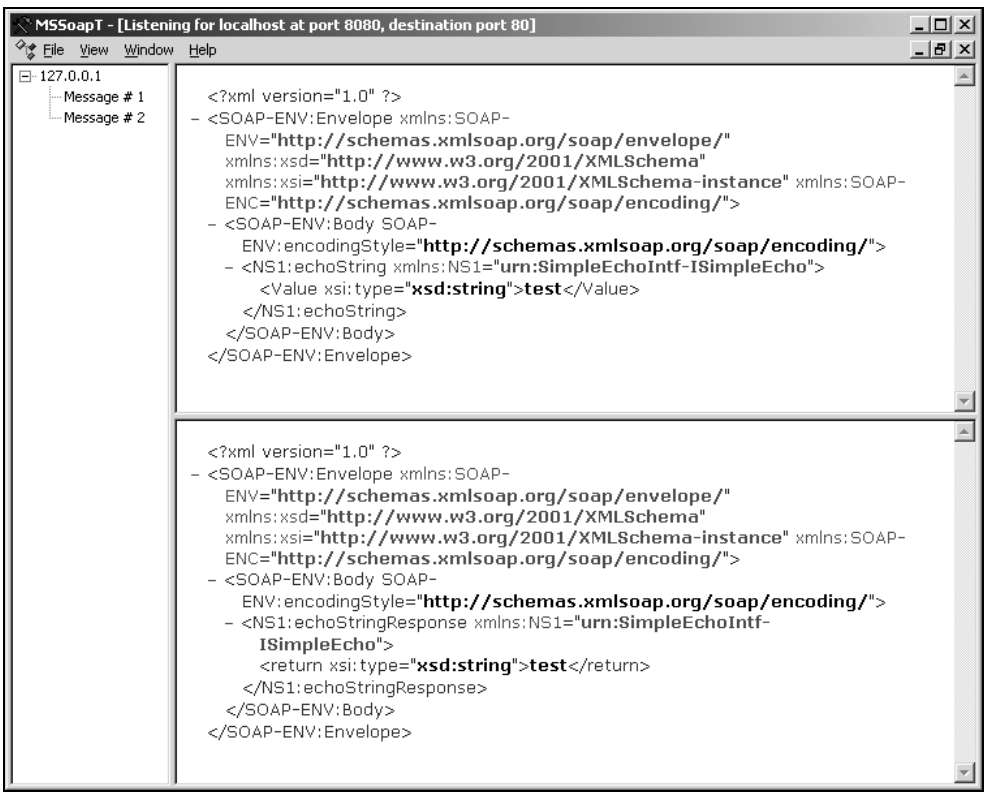


Рис. 18.8. Утилита SOAP Trace — просмотр в режиме форматированного вывода

Во-вторых, при вводе параметров утилиты в поле **Destination host** нужно занести истинный адрес сервера — **soap.myservices.com**. Обращаем ваше внимание — 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. 18.8. 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, которые легли в основу 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 особое внимание.

Глава 19

Технология WebSnap

Большинство современных распределенных приложений содержат части, доступ к которым осуществляется через Internet. Естественно, такие приложения используют для представления данных и взаимодействия с пользователями возможности Web-технологий. Для разработки такого рода функциональности распределенных приложений в состав Delphi включена технология *WebSnap*. Она наследует, развивает и существенно расширяет возможности другой технологии Delphi для работы с приложениями, Web-технологии *Web Broker*, которая применялась и в более ранних версиях Delphi.

Приложения WebSnap используют 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.

В этой главе рассматриваются следующие вопросы:

- состав приложения WebSnap и использование стандартных мастеров WebSnap;
- назначение модулей данных и модулей страниц;
- роль компонента `Tadapter`;
- диспетчеризация запросов клиентов;
- разработка интерфейса и бизнес-логики приложения WebSnap;
- использование шаблонов.

19.1. 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 дальше) 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.

Первая часть обеспечивает управление потоками запросов и ответов. Ее 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 данным и хранение бизнес-логики приложения.

Рассмотрим перечисленные составные части подробнее (рис. 19.1).

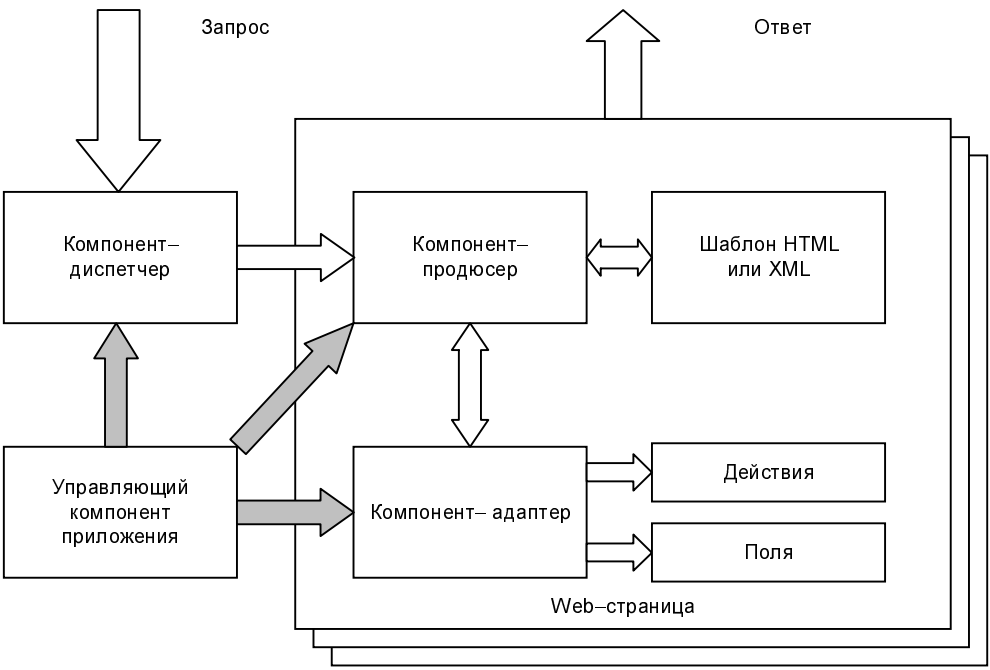


Рис. 19.1. Структура приложения WebSnap

Сначала запрос поступает *компоненту-диспетчеру* `TPageDispatcher`, которому "известно" назначение *страниц* приложения, он направляет им запрос для дальнейшей обработки и формирования ответа. При этом модуль страницы обеспечивает адекватную реакцию страницы на запрос.

На этапе создания приложения для каждой его страницы формируется *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 использовать скрипты на языках VBScript или JScript.

Во время выполнения приложения запрос поступает *компоненту-продюсеру*, связанному со страницей. Он обеспечивает генерацию страницы на основе имеющегося шаблона.

Примечание

Разнообразные компоненты-продюсеры размещены на странице **Internet** Палитры компонентов Delphi.

Если же шаблон содержит скрипт, то его выполнение обеспечивается *компонентами-адаптерами*. В зависимости от выполняемых задач разработчик может использовать несколько типов компонентов-адаптеров, расположенных на странице **WebSnap** Палитры компонентов. Для направления запросов нужным адаптерам должен применяться 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 осуществляется компонентом `TWebAppComponents`. Он объединяет части приложения WebSnap в единую "технологическую цепочку", а разработчик может динамически заменять звенья цепи (перечисленные выше компоненты) в зависимости от потребностей бизнес-логики приложения. Для этого используются свойства компонента, ссылающиеся на соответствующие компоненты: адаптеры и диспетчеры.

19.1.1. 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. Это может быть адаптер набора

данных (компонент `TDataSetAdapter`), который, к примеру, обеспечит получение записей набора данных из базы данных.

19.1.2. Web-модули

С точки зрения разработчика, приложение WebSnap в первую очередь состоит из специализированных модулей, в которых можно размещать необходимые компоненты. Эти Web-модули делятся на два вида (рис. 19.2): модули страниц и модули данных.

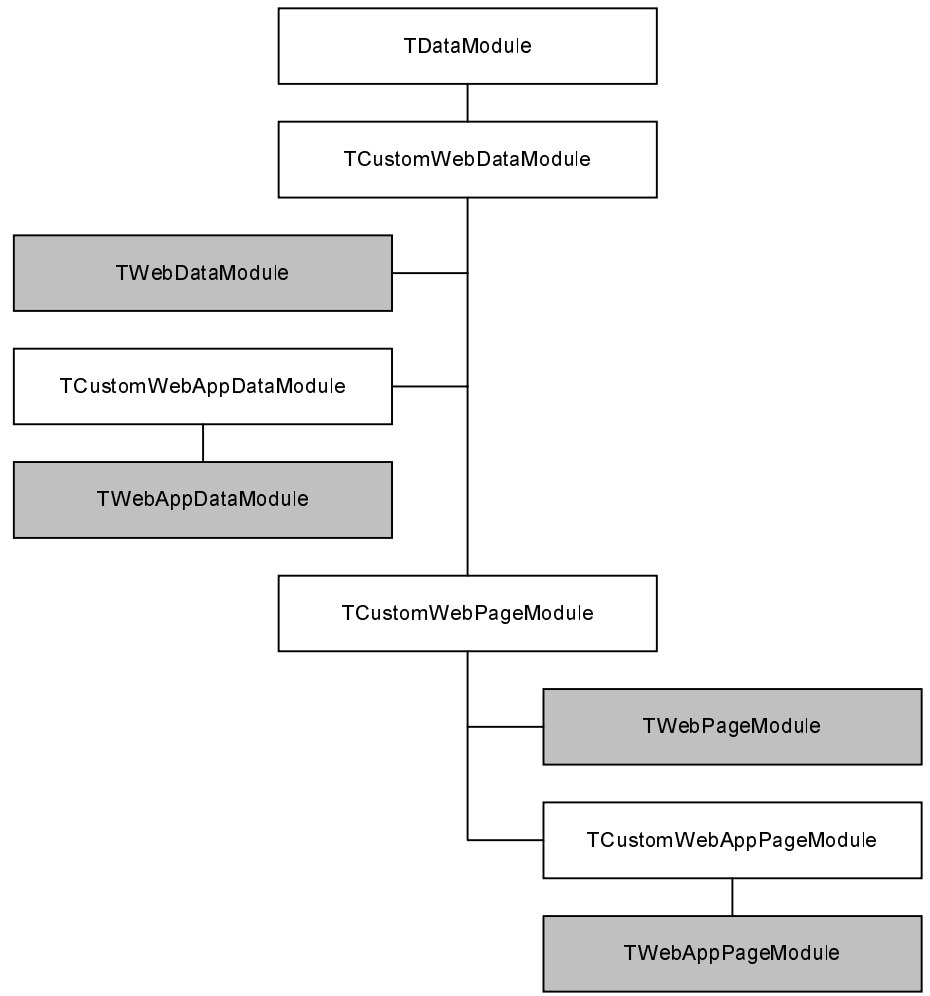


Рис. 19.2. Иерархия классов Web-модулей

Модули страниц HTML реализуют пользовательский интерфейс приложения. Они выполняют две функции:

- содержат компонент-продюсер, обеспечивающий динамическую генерацию страницы на основе шаблона HTML или XML;
- хранят шаблон страницы.

Модули данных являются просто платформами для размещения любых невизуальных компонентов и аналогичны модулям данных, используемым в приложениях баз данных *(см. гл. 5)*.

Кроме этого, Web-модули могут быть *модулями приложения*. Такие модули предоставляют приложению WebSnap хранилище для компонентов, реализующих важнейшие функции приложения в целом. Приложение может иметь только один такой Web-модуль. В качестве модуля приложения может выступать как модуль страницы, так и модуль данных.

Общим предком любого Web-модуля является класс `TDataModule`.

Модуль данных

Модуль данных приложения WebSnap (подобно обычному модулю данных) является платформой для любых невизуальных компонентов. В нем можно размещать как компоненты управления приложением WebSnap, так и, к примеру, компоненты доступа к данным.

Функциональность модуля данных описана 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, обращенный 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.

Кроме того, из модуля данных можно управлять сеансом доступа клиента. Для этого предназначено свойство

```
property Session: TAbstractWebSession;
```

Абстрактный класс сессии `TAbstractWebSession` позволяет своим потомкам реализовать 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), в котором хранится 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. 19.3). Дополнительные вкладки, расположенные справа внизу, позволяют редактировать не только шаблон HTML для страницы (вкладка **HTML Result**), но и просматривать внешний вид страницы без компиляции приложения, как будто приложение выполняется (вкладка **Preview**). Эта функция дает возможность очень быстро и легко создать и настроить пользовательский интерфейс страницы.

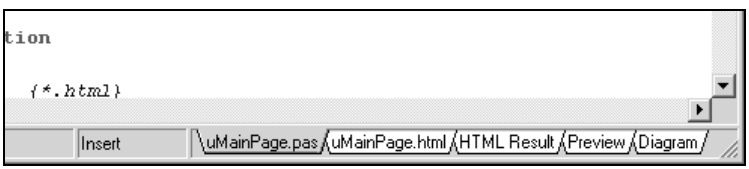


Рис. 19.3. Окно Редактора кода Delphi для модуля страницы

Класс `TWebPageModule` реализует функции Web-страницы.

На каждом модуле страницы должен быть размещен компонент-продюсер `TPageProducer`. Это обязательный компонент, так как он обеспечивает генерацию страницы из шаблона на этапе выполнения приложения. Для связывания компонента и модуля страницы используется свойство модуля

```
property PageProducer: IProduceContent;
```

которое должно ссылаться на компонент-продюсер. В общем случае страница приложения может обладать сложными функциями, которые реализуются при помощи различных компонентов-продюсеров. Например, это может быть обычный компонент `TPageProducer` и компонент-продюсер `TDataSetPageProducer` для организации доступа к набору данных какой-либо базы данных. В этом случае у разработчика есть возможность переназначать свойство `PageProducer` модуля страницы динамически во время выполнения.

Свойство

```
property DefaultAction: TAdapterActionName;
```

применяется для связывания модуля 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 свойствам класс модуля страницы имеет и обычные для Web-модуля свойства — ведь он, кроме того, обеспечивает управление запросами клиентов. В нем также 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;
```

Более подробную информацию об этих свойствах см. в статье "Модуль данных" данного раздела.

Модуль приложения

Базовая функциональность приложения WebSnap обеспечивается модулем приложения. Этот модуль должен содержать компоненты, инкапсулирующие функции общего управления приложением и обработку запросов.

У приложения WebSnap всегда 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 осуществлять общее управление.

Класс `TWebAppDataModule` является модулем данных приложения. Он не входит в пользовательский интерфейс приложения и не доступен во время выполнения.

Любой модуль приложения наследует свойства и методы от соответствующего класса обычного модуля страницы или модуля данных (см. рис. 19.2) и обладает всеми их возможностями для обработки 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 в нем размещаются компоненты-диспетчеры страниц и адаптеры.

Примечание

Если модуль приложения является модулем Web-страницы, то при его разработке дополнительно создается и компонент `TPageProducer`, как для обычного модуля страницы.

19.1.3. Компоненты уровня приложения

Помимо специализированных компонентов WebSnap, приложение может использовать 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 с запросами;
- хранят информацию 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, используется компонент `TwebAppComponents` (рис. 19.4). Он имеет ряд свойств, ссылающихся на другие нужные приложению компоненты. Разработчик имеет возможность корректировать функциональность приложения, заменяя необходимые компоненты на этапе выполнения.

Компонент `TWebAppComponents` размещается в модуле приложения.

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.

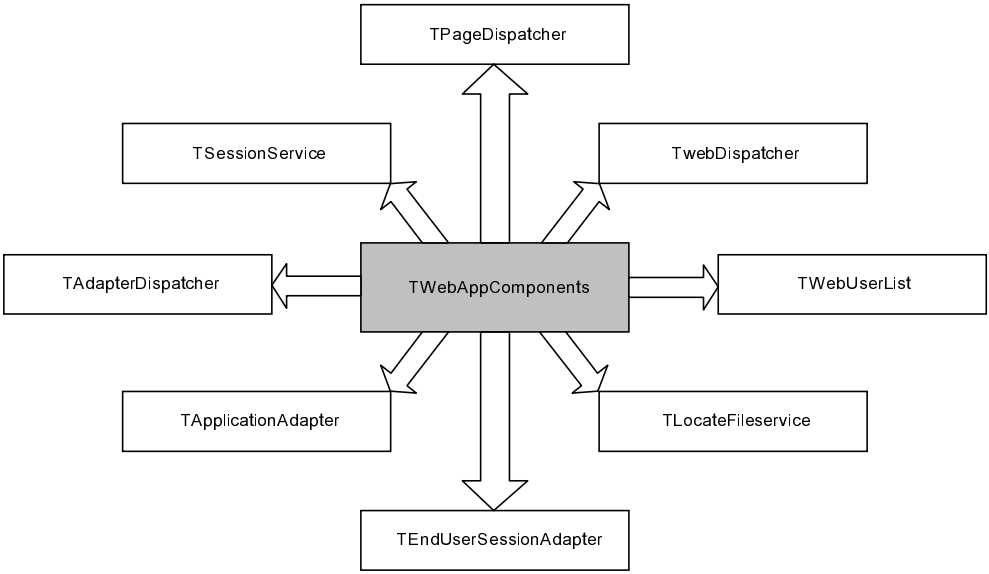


Рис. 19.4. Управление взаимодействием компонентов приложения WebSnap с помощью компонента `TWebAppComponents`

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, используемые 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 в многопользовательском режиме.

Список всех пользователей приложения и их пароли могут храниться в компоненте `TWebUserList`. А свойство

```
property UserListService: IWebUserList;
```

позволяет подключить этот компонент для использования приложением.

Параметры текущего пользователя приложения сохраняются в компоненте `TWebEndUser`, а сам он связывается с приложением с помощью свойства

```
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 содержит 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

Приложение может использовать собственный набор действий, не связанный с каким-либо из компонентов-адаптеров модулей страниц или данных. Для этого используется 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 во время работы приложения обеспечит отображение общего заголовка приложения для каждой страницы.

В приведенном примере используется поле `Title`. Это поле создается для компонента по умолчанию и отображает значение опубликованного (published) свойства компонента

```
property ApplicationTitle: string;
```

В дальнейшей работе вы можете самостоятельно создать необходимые поля и действия и применять их на всех страницах приложения.

Информация о пользователях

Приложение WebSnap сохраняет и позволяет использовать информацию о своих пользователях. Для этих целей предназначены 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 используются только для программирования дополнительных операций. *Более подробную информацию 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 на то, что здесь вы можете конкретизировать реакцию вашего приложения на ошибку, используя параметр `TValidateUserError`:

```
TValidateUserError = (vuBlankPassword, vuBlankUserName, vuUnknownUser-
Name, vuUnknownPassword);
```

Например, в простейшем случае это можно сделать так:

```
procedure TSomePage.WebUserList1ValidateUserError(Error: TValidateUserEr-
ror; 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`. Компонент позволяет использовать глобальную переменную `EndUser` в скриптах страниц приложения.

Как и любой адаптер, компонент `TEndUserAdapter` содержит индексированные списки действий и полей в стандартных свойствах:

```
property Actions: TAdapterDefaultActions;
```

и

```
property Data: TAdapterDefaultFields;
```

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 отображения используется метод:

```
EndUser.LoginForm.Enabled
```

А для указания имени страницы регистрации используется свойство

```
property LoginPage: string;
```

Действие `LogOut` отключает текущего пользователя.

Поле логического типа `LoggedIn` позволяет проверить, прошел пользователь аутентификацию или нет. А поле `DisplayName` хранит имя пользователя.

Сессии

К приложению WebSnap одновременно могут обращаться несколько пользователей. Следовательно, 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 связывается 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: TAbstractWeb-
Session) of object;
property OnStartSession: TStartSessionEvent;
```

и

```
type
  TEndSessionEvent = procedure(ASender: TObject; ASession: TAbstractWeb-
Session; 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 используется 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 компоненту 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 применением этих файлов предназначен 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.

19.1.4. Компоненты-диспетчеры

Компоненты-диспетчеры играют важную роль в приложениях WebSnap. Они управляют 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 доступом пользователей к странице. Для этого используется 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 — адресата запроса, вызывается метод-обработчик

```
property OnPageNotFound: TDispatchPageHandledEvent;
```

Компонент `TAdapterDispatcher` связывает 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 программисту отслеживать вызов полей или действий компонентов-адаптеров.

19.1.5. 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 используется 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 достаточно использовать метод

```
function Content: string;
```

но при этом обязательно должен быть задан шаблон страницы при помощи свойств `HTMLDoc` или `HTMLFile`. В противном случае можно использовать 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** находятся еще четыре специализированных компонента-продюсера, которые обеспечивают загрузку данных для генерируемой страницы из связанного набора данных.

19.1.6. 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. 19.5).

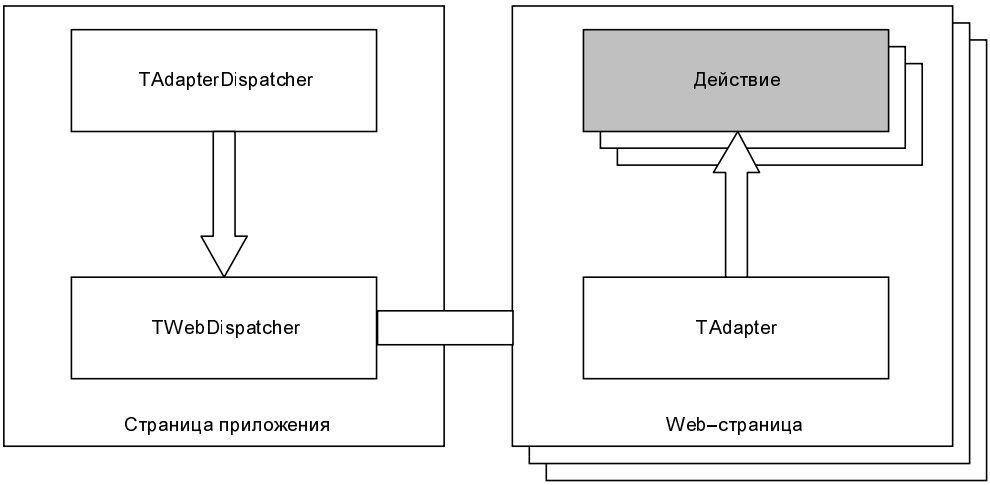


Рис. 19.5. Механизм вызова действий компонентов-адаптеров

Поля компонента-адаптера хранят переменные, необходимые при работе страниц приложений и выполнении действий.

Наиболее универсальным компонентом-адаптером из имеющихся в распоряжении технологии WebSnap является компонент `TAdapter`.

Индексированный список действий адаптера содержится в свойстве

```
property Actions: TAdapterActions;
```

При щелчке на кнопке свойства в Инспекторе объектов появляется окно Редактора действий (рис. 19.6). 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 в приложении. Здесь разработчик может предусмотреть выполнение любых необходимых операций. Параметр `Params` позволяет использовать необходимые переменные действий.

Рис. 19.6. 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 содержится в свойстве

```
property Data: TAdapterFields;
```

При щелчке на кнопке свойства в Инспекторе объектов появляется Редактор, аналогичный Редактору действий (см. рис. 19.6). При создании нового объекта поля разработчик имеет возможность выбрать тип данных поля из списка. Описание возможных типов полей представлено в табл. 19.1.

Отдельное поле описывается базовым классом `TAdapterField`. Другие типы полей, представленные в табл. 19.1, имеют собственные классы, которые наследуют основные возможности от базового класса. Их основное отличие — реализация методов-обработчиков для получения и задания значений в соответствии с поддерживаемыми типами данных.

Таблица 19.1. *Типы полей для компонента-адаптера* `TAdapter`

Класс	Описание
`TAdapterField`	Универсальное поле для представления простых и строковых типов данных
`TAdapterBooleanField`	Специализированное поле для хранения переменных логического типа
`TAdapterFileField`	Поле для хранения имен файлов, использующихся при работе приложения WebSnap

Таблица 19.1 *(окончание)*

Класс	Описание
`TAdapterImageField`	Поле для представления бинарных массивов
`TAdapterMemoField`	Поле для представления данных типа Memo
`TAdapterMultiValueField`	Поле для сохранения нескольких значений одновременно или для хранения одномерных массивов. Доступ к отдельным значениям обеспечивается параметром `Index`

Теперь 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 на то, что использование этих методов соответствует способу работы со свойствами в классах Delphi. 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 использоваться при создании элементов управления страниц (например, списков или радиогрупп). Для этого применяется специализированный компонент-адаптер `TStringsValuesList`. Он имеет свойство

```
property Strings: TStrings;
```

в котором в соответствии с форматом: `Name=Value` задаются необходимые значения.

В компоненте `TAdapter` ссылка на компонент со списком значений содержится в свойстве

```
property ValuesList: TComponent;
```

Более подробно об использовании компонентов-адаптеров в приложениях WebSnap рассказывается далее в этой главе.

19.2. Создание приложения WebSnap

Для разработки приложения WebSnap и его составных частей — модулей страниц и модулей 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. 19.7.

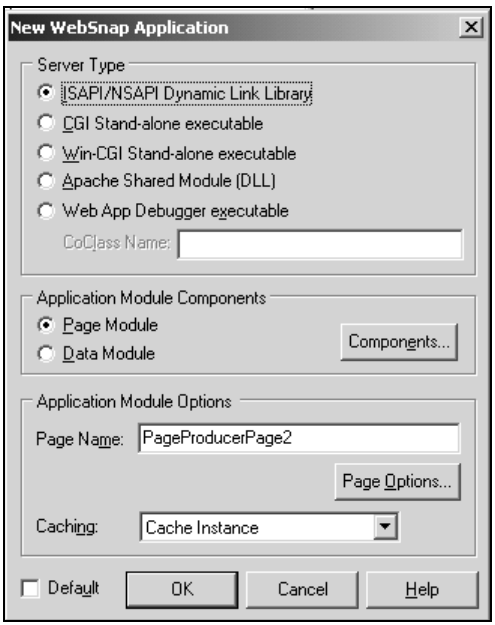


Рис. 19.7. Мастер создания приложения WebSnap

Группа радиокнопок **Server Type** предназначена для выбора типа создаваемого приложения:

- **ISAPI/NSAPI Dynamic Link Library** — динамическая библиотека, работающая в адресном пространстве сервера Internet Information Server в многопользовательском режиме;
- **CGI standalone executable** — исполняемый файл CGI, получающий управляющие команды из переменных окружения;
- **Win-CGI standalone executable** — разновидность CGI, адаптированная для использования в Windows;
- **Apache Shared Module (DLL)** — динамическая библиотека, работающая в адресном пространстве сервера Apache под управлением ОС UNIX в однопользовательском режиме;

- **Web App Debugger executable** — исполняемый файл, позволяющий отлаживать приложение средствами Delphi без использования реальных Web-серверов.

В процессе 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.

Здесь же располагается кнопка **Components**, при нажатии на которую разворачивается окно **Web App Components** (рис. 19.8). В нем можно выбрать компоненты WebSnap, которые будут размещены в модуле приложения. По умолчанию выделены три компонента: `TApplicationAdapter`, `TPageDispatcher`, `TAdapterDispatcher`. Их назначение вам уже известно.

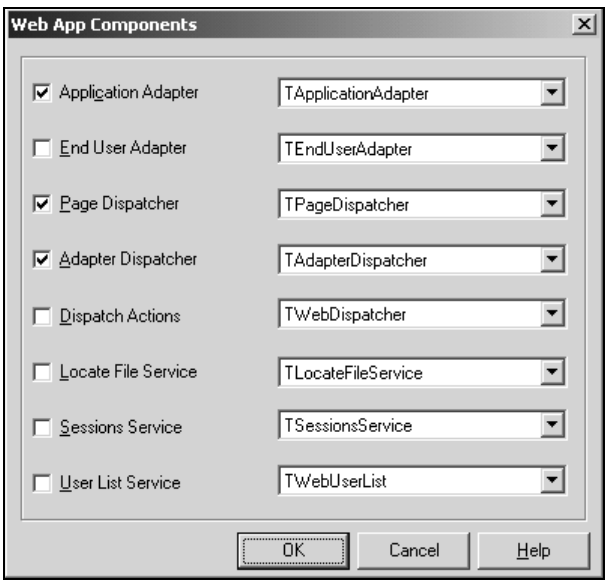


Рис. 19.8. Окно **Web App Components** выбора компонентов для Web-модуля

Примечание

Кроме перечисленных компонентов мастер обязательно включает в приложение компонент `TWebAppComponents`.

Затем в строке ввода **Page Name** (см. рис. 19.7) следует задать имя создаваемой страницы вместо предложенного по умолчанию.

Список **Caching** позволяет настроить поведение создаваемого модуля на этапе выполнения приложения, после обработки запроса клиента. Если модуль не имеет большого значения или используется редко, можно задать значение

Destroy Instance. В этом случае сразу после выполнения запроса модуль удаляется из памяти. Значение **Cache Instance** сохранит модуль в памяти.

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. 19.9).

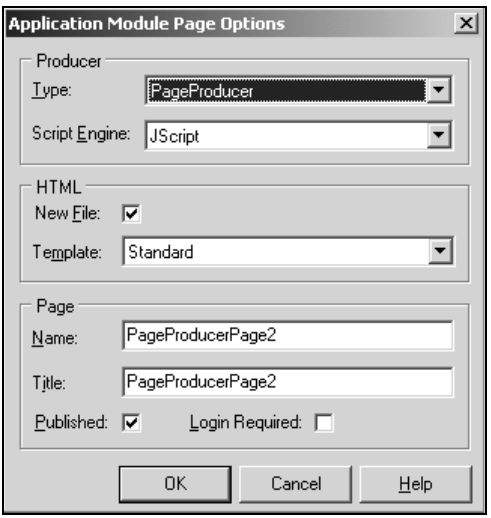


Рис. 19.9. Окно **Application Module Page Options** для дополнительной настройки создаваемого Web-модуля

Здесь в списке **Type** группы **Producer** можно выбрать компонент-продюсер, который будет использоваться для генерации страницы во время выполнения приложения. Если для создаваемой страницы планируется применение скриптов, 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** (см. рис. 19.9). В ней задаются параметры соответствующего шаблона.

Установленный 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).

Затем в списке **Template** нужно задать тип шаблона. По умолчанию выбрано значение **Standard**, и файл шаблона будет заполнен в соответствии с выбранным языком скрипта. Значение **Blank** запрещает генерацию исходного шаблона.

Значение **Data Packet** используется только для шаблонов XSL и генерирует соответствующий шаблон.

В строках **Name** и **Title** задаются имя и заголовок страницы соответственно.

Установленный флажок **Published** позволяет странице автоматически отвечать на запрос, если запрос адресован странице напрямую.

Флажок **Login Reqired** включает авторизацию пользователей при доступе к странице.

После настройки всех параметров будущего приложения и нажатия кнопки **OK** будет созданы исходные файлы нового проекта Delphi и модуля приложения.

Для доступа к файлам модуля приложения (PAS, HTML, XML или XSL) и любых других Web-модулей используются вкладки 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 описаны. Единственный дополнительный список выбора **Creation** позволяет выбрать способ формирования страницы. При значении **On Demand** страница будет создана только при первом обращении к ней. Значение **Always** задает обязательное создание страницы.

При разработке модуля данных необходимо задать только способ его создания и кэширования.

19.3. Конструирование интерфейса и управление данными

Теперь, имея представление о структуре приложения WebSnap, перейдем к небольшому практикуму и рассмотрим некоторые особенности применения компонентов WebSnap.

Воспользовавшись Мастером создания приложения WebSnap, создадим новое приложение. В качестве типа сервера (см. рис. 19.7) 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'`. Учтите, 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 готово к выполнению и отладке. Для этого необходимо запустить утилиту WebApp Debugger (рис. 19.10), выбрав одноименный пункт из меню **Tools** Delphi.

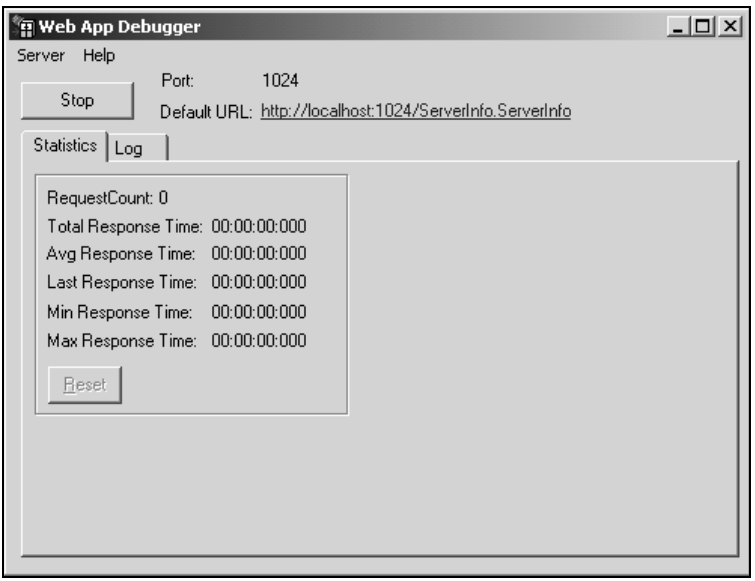


Рис. 19.10. Утилита Web App Debugger для отладки приложений WebSnap

Если сервер еще не запущен, это необходимо сделать при помощи кнопки **Start**. Затем выберите ссылку **Default URL** и в развернувшемся окне Internet Explorer — ссылку **ViewDetails**. В появившемся списке укажите и запустите тестовое приложение — появится главная страница приложения **MainPage**.

По аналогичной схеме мы будем тестировать все последующие этапы разработки.

Вопросы создания пользовательского интерфейса на основе Web-страниц выходят за рамки нашего обсуждения. Заинтересованные читатели могут обратиться к разнообразной литературе на эту тему. Здесь же отметим, что в окне Редактора Delphi есть возможность 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.

19.3.1. 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 19.1).

Листинг 19.1. 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 `e`, содержащий страницы приложения, доступные для просмотра. Затем в цикле в строку `s` добавляются имена страниц, и если счетчик страниц `c` содержит больше одной страницы, панель навигации по страницам отображается в главной странице приложения. Для этого используется метод `Response.Write(s)`, где `Response` — ссылка на объект ответа на запрос.

19.3.2. Использование полей и действий компонентов-адаптеров

Компоненты-адаптеры WebSnap предоставляют разработчику два важнейших инструмента, применяемых для создания 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 использования этих возможностей попробуем создать простую страницу HTML для просмотра текстовых файлов. К созданному выше приложению добавим новую страницу `TextFileViewer`.

Шаблон этой страницы создан при помощи программы Microsoft Front Page и содержит однострочный редактор 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 добавим новое поле. Для этого необходимо щелкнуть на кнопке свойства `Data` в Инспекторе объектов и в появившемся Редакторе полей создать новое поле. Тип поля — `TAdapterFileField`. Это поле предназначено для хранения текущего имени файла. Затем добавим еще одно поле, выбрав для него тип `TAdapterMemoField`. Это поле используем для представления содержимого выбранного файла.

Для новых полей создадим методы-обработчики события `OnGetValue`, как это показано в листинге 19.2.

Поле `FileName` возвращает полное имя текущего файла, а поле `FileContent` возвращает содержимое этого файла в формате `Memo`. При этом его метод-обработчик `FileContentGetValue`, используя значение переменной `FFileName`, создает объект файлового потока и заполняет переменную `FFileContent`.

Теперь необходимо описать 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. 19.6). Добавим в нем новый объект класса `TadapterAction` и назовем его `SelectFileName`. Затем создадим метод-обработчик `OnExecute` и поместим в него код для вызова стандартного диалогового окна открытия файла (см. листинг 19.2).

Листинг 19.2. 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
WebRequestHan-
dler.AddWebModuleFactory(TWebPageModuleFactory.Create(TTextFileViewer,
TWebPageInfo.Create([wpPublished {, wpLoginRequired}], '.html'), crOnDe-
mand, 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 `On-Execute` действия `SelectFileName`. Для этого в шаблон страницы следует включить следующие объявления:

```
<input type=hidden name="__action"
value="<%=FileViewerAdapter.SelectFileName.AsFieldValue%>">
<input type="submit" value="Open" name="buOpen" onclick="F.__action.value=
'<%=FileViewerAdapter.SelectFileName.AsFieldValue%>'">
```

Здесь событие нажатия кнопки связывается с выполнением действия `SelectFileName`, то есть 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.

После компиляции и запуска приложения страница **TextFileViewer** позволяет при нажатии кнопки **Open** выбирать в стандартном диалоговом окне файлы и просматривать их содержимое в многострочном редакторе. При этом имя файла отображается в однострочном редакторе.

19.3.3. 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 описанному ранее приложению SimpleWebSnap Application добавим новую страницу и назовем ее **DataViewer**. В этом случае будут использованы другие компонент-адаптер и компонент-продюсер. Поскольку в простейшем случае удобнее всего отображать данные в табличном виде, вместо стандартного компонента-продюсера `TPageProducer` используем компонент `TAdapterPageProducer`, а вместо славно потрудившегося в предыдущих примерах компонента `TAdapter` применим `TDataSetAdapter`. И теперь давайте посмотрим, какие же преимущества дает использование новых компонентов.

Специализированные элементы управления

Компонент `TAdapterPageProducer` предоставляет дополнительные возможности для создания Web-страниц и связывания элементов управления этих страниц с полями и действиями компонента-адаптера. Для этого необходимо нажать кнопку в Инспекторе объектов в строке свойства компонента

```
property WebPageItems: TWebComponentList;
```

Появится окно Редактора свойства, представленное на рис. 19.11.

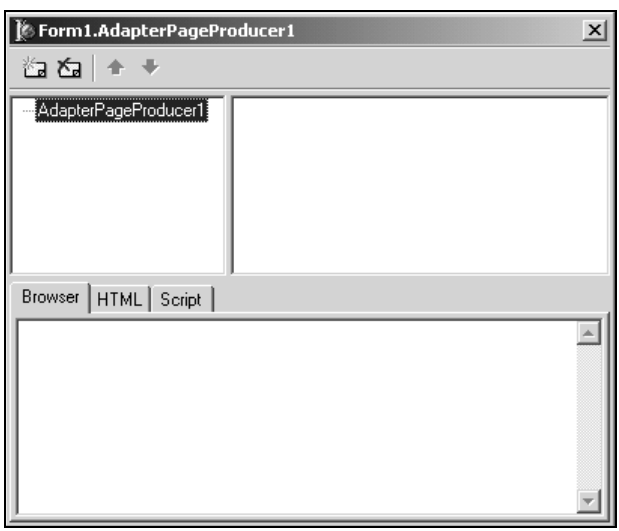


Рис. 19.11. Окно Редактора компонента `TAdapterPageProducer`

В верхней части окна расположен иерархический список, подобный применяемому в редакторам полей и действий компонентов-адаптеров

(см. рис. 19.6). 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.

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.

На форме HTML для компонента `TAdapterPageProducer` доступны элементы управления следующих классов.

Класс `TAdapterFieldGroup` обеспечивает представление полей связанного набора данных в виде текстового списка имен и значений полей. Одновременно 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 на той же форме.

Класс `TAdapterGrid` описывает 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 не доступен в иерархическом дереве редактора и отображается после создания только в списке в правой части окна. Он предназначен для обработки возможных исключительных ситуаций наборов данных.

Просмотр данных

В нашем примере достаточно отобразить данные в табличной форме. Для этого в Редакторе компонента `TAdapterPageProducer` добавим форму HTML (экземпляр класса `TAdapterForm`) и на ней — таблицу (экземпляр класса `TAdapterGrid`). Результат наших несложных приготовлений можно увидеть здесь же. Для этого достаточно выбрать вкладку **Browser** в нижней части окна редактора, текст шаблона доступен на вкладке **Script**. Однако на этом этапе мы не видим ни данных, ни таблицы. Вместо этого отображается сообщение об ошибке (рис. 19.12).

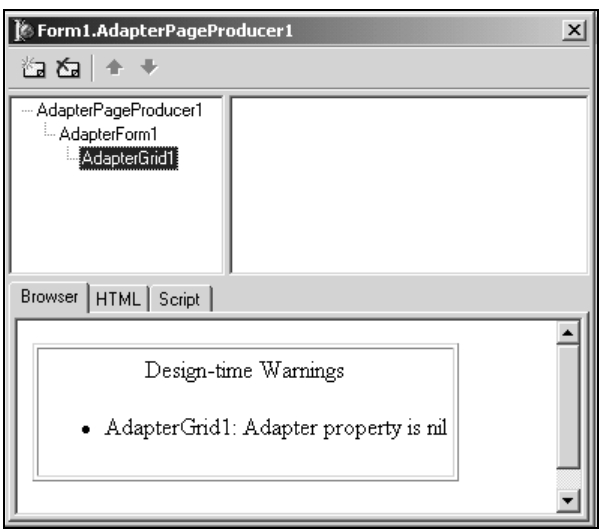


Рис. 19.12. Сообщение об ошибке на вкладке **Browser** Редактора компонента `TAdapterPageProducer`

Тем не менее, все идет по плану. Дело в том, что мы не связали табличный элемент управления с набором данных.

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 уже упоминавшееся свойство `Adapter`, которое позволяет связать их с компонентом-адаптером. В нашем случае это должен быть компонент `TDataSetAdapter`, так как он "умеет" взаимодействовать с компонентами наборов данных. Для этого предназначено его свойство

```
property DataSet: TDataSet;
```

Поскольку оно имеет тип `TDataSet`, адаптер может работать с компонентами различных технологий доступа к данным *(см. часть 2 "Технологии доступа к данным")*. В нашем примере используем 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. *(Более подробную информацию о настройке этих компонентов см. в гл. 7.)*

Итак, после выполнения всех перечисленных операций и соединения элемента управления `TAdapterGrid` с адаптером `TdataSetAdapter`, а через него с компонентом `TADOTable`, страница начинает работать корректно, и на вкладке **Browser** Редактора компонента `TAdapterPageProducer` появляются данные из таблицы `Country` базы данных.

Аналогичным образом можно использовать и другие элементы управления.

Редактирование данных

Для редактирования данных из таблицы `Country` добавим в иерархический список элементов управления компонента `TAdapterPageProducer` еще одну форму и на 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 компонентом-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. 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`.

19.3.4. 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 ему присваивается значение `True`, то исключительная ситуация будет обработана в этом методе-обработчике.

Исключительные ситуации на уровне отдельных страниц можно обрабатывать при помощи свойства компонентов-адаптеров

```
property Errors: TBaseAdapterErrorsList;
```

Это свойство предоставляет разработчику 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, использовав метод

```
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;
```

19.4. Аутентификация пользователей

В Web-приложениях часто бывает необходимо обеспечить аутентификацию пользователей при подключении или обращении к определенным страницам. В технологии WebSnap для этого используются описанные ранее специализированные компоненты.

В качестве примера добавим к созданному выше тестовому приложению возможность аутентификации пользователей. Для этого необходимо:

- интегрировать в приложение список имен пользователей и паролей или добавить функцию, позволяющую получать такой список из внешних источников (например, базы данных или зашифрованного файла);
- создать страницу, содержащую элементы управления для ввода имени и пароля пользователя;
- обеспечить взаимодействие компонентов и страниц приложения;
- выделить страницы приложения, доступ к которым осуществляется только авторизованными пользователями, и настроить их.

Рассмотрим эти этапы более подробно.

19.4.1. Списки пользователей и сессии

Процесс аутентификации пользователя подразумевает, что приложение сможет сравнить введенные имя пользователя и пароль с имеющимися в списке пользователей. Такой список содержит компонент `TWebUserList`. В приложении SimpleWebSnap Application этот компонент необходимо перенести в модуль приложения. При щелчке на кнопке свойства `UserItems` в Инспекторе объектов появляется диалоговое окно, позволяющее 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 помещается в компонент `TEndUserSessionAdapter`. Его также необходимо перенести в модуль приложения.

Обратите внимание на то, что после переноса в модуль страницы все три компонента автоматически связываются с компонентом `TWebAppComponents` через соответствующие свойства. Тем самым обеспечивается единое управление всеми компонентами для аутентификации пользователей.

19.4.2. Создание страницы аутентификации

Страница аутентификации — это обычная Web-страница, обеспечивающая ввод имени и пароля пользователя. Для приложения SimpleWebSnap Application создадим еще одну страницу HTML, выбрав для нее компонент-продюсер `TAdapterPageProducer`, подобно уже описанной странице, связанной с базой данных. Имя новой страницы — `UserLogin`.

В этом случае существует маленькое отличие — страница не должна быть доступной на этапе создания приложения. Для этого необходимо сбросить флажок **Published** в окне мастера создания страницы (см. рис. 19.9).

С помощью 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 на вкладке **Browser** можно увидеть созданную страницу (рис. 19.13).

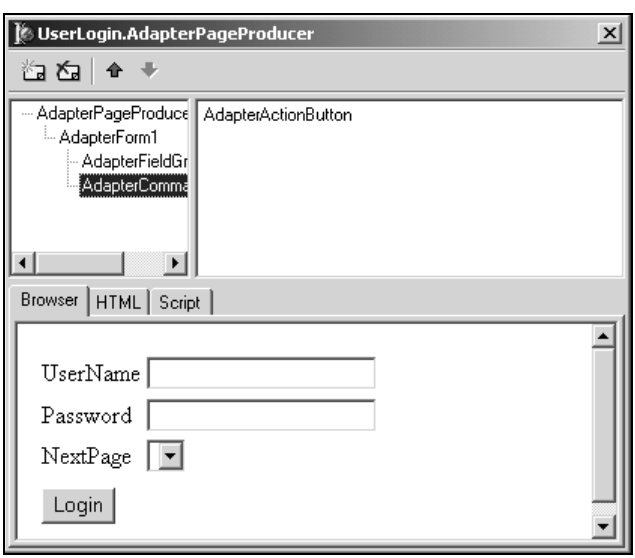


Рис. 19.13. Стандартная страница аутентификации пользователей приложения WebSnap

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 случится, но на наш взгляд они все же лишние. Поэтому шаблон страницы аутентификации в тестовом приложении выглядит следующим образом (листинг 19.3).

Листинг 19.3. Файл HTML-страницы аутентификации

```
<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.

19.4.3. Настройка приложения

И в завершение необходимо связать созданную страницу с процессом аутентификации в приложении и определить, какие страницы будут защищены паролем.

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 выборе ссылки `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 символы комментарияи у параметра `wpLoginRequired`.

Теперь во время выполнения приложения при попытке открыть эту страницу автоматически будет вызываться страница с предложением ввести имя и пароль пользователя.

Проверка созданного механизма аутентификации имеет особенность. При аутентификации пользователя используется информация о сессии приложения. А она сохраняется, только если приложение работает все время, без завершения выполнения между вызовами страниц.

Поскольку 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.

19.5. Использование XML и XSL

Наряду с шаблонами HTML приложения WebSnap могут использовать и шаблоны XML. Для этого используется 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 использования XML и XSL технологией WebSnap. *Более подробную информацию об XML и XSL см. в гл. 15.*

Для демонстрации возможностей применения XML создадим новое приложение Simple XML WebSnap Application, подобно уже описанному 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. Поскольку приложение должно использовать шаблоны XML, при создании страницы в мастере (см. рис. 19.9) необходимо выбрать компонент-продюсер `TXSLPageProducer`.

Далее мы расскажем, как обеспечить генерацию страниц на основе файлов XML и XSL или информации, получаемой из базы данных.

19.5.1. Использование файла XML и шаблона XSL

Сначала посмотрим, как можно применять 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 вкладка для файла XSL.

Настройка страницы для использования 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 в них 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;
- или использовать для указания пути к файлу URL.

Примечание

Файл SimpleXMLDataList.xml создан с применением стандарта Microsoft XML Integration in ADO 2.5, описывающего экспорт и импорт 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 к странице необходимо присоединить таблицу стилей XSL.

Для этого необходимо перенести в папку приложения нужный файл XSL и переименовать его в соответствии с именем файла создаваемой страницы.

В нашем случае это должен быть файл uListViewer.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 обеспечивает простейший способ отображения данных — в две текстовые колонки (листинг 19.4).

Листинг 19.4. 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, 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 значений полей.

19.5.2. Использование набора данных и шаблона XSL

Кроме файлов XML страницы на основе шаблонов XML и XSL можно создавать и для данных, хранящихся в базах данных.

Для демонстрации этой возможности технологии WebSnap создадим еще одну страницу приложения Simple XML WebSnap Application. Назовем ее **DataViewer**. Так же 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 в нем компоненты ADO: `TADOConnection` и `TADOTable`. Настроим их для доступа к базе данных Microsoft Access DBDEMOS.MDB из демонстрационной базы данных Delphi. *(Более подробную информацию о технологии и компонентах ADO см. в гл. 7.)* Кроме того, нам понадобится компонент `TDataSetProvider`, который обеспечит передачу данных. *(Подробнее роль компонента `TDataSetProvider` описана в гл. 8.)* Свяжем между собой компонент таблицы ADO и компонент `TDataSetProvider`, используя его свойство `DataSet`.

Затем перенесем в модуль данных создаваемой страницы **DataViewer** компонент `TXMLBroker` со страницы **Internet Express** Палитры компонентов. 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 на страницу XML, настроена. Осталось создать стиль XSL. Это делается так же, как и для предыдущего случая. Сам файл XSL этой страницы описывает таблицу для представления данных (листинг 19.5).

Листинг 19.5. 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, 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">
```

19.5.3. Дополнительный компонент *TAdapterXMLBuilder*

Кроме стандартных компонентов WebSnap, о которых уже много рассказывалось в этой главе, для создания страниц XML разработчики могут использовать 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 компонентов компонент появляется в списке выбора продюсеров в Мастере создания Web-страниц (см. рис. 19.9).

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. 19.14).

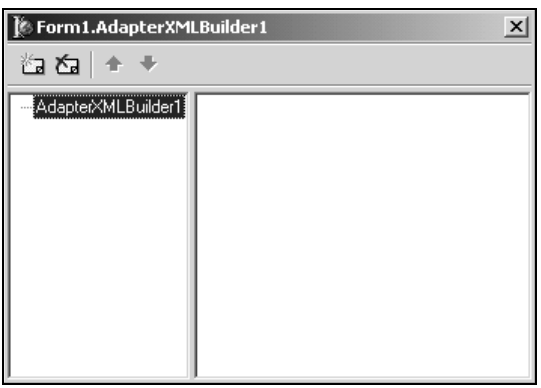


Рис. 19.14. Окно Редактора компонента TAdapterXMLBuilder

Элементы управления можно располагать только на форме класса 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 нарямую, а с помощью 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 на вкладке **XML Tree** окна Редактора Delphi.

Резюме

Технология WebSnap обеспечивает создание широкого спектра Web-приложений. Эти приложения могут быть следующих типов:

- ISAPI/NSAPI DLL;
- CGI;
- Win-CGI;
- Apache Shared Module (DLL);
- Web App Debugger.

Обработку запросов, поступающих приложению WebSnap от Web-сервера, обеспечивает триада 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.

Для реализации сложных функций 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 дискеты

Уважаемый читатель!

Ниже приводится описание примеров, приведенных в этой книге. Каждый пример хранится в отдельной папке и представляет собой полноценный проект, который можно загружать в среду разработки и компилировать.

Название каталогов состоит из номера главы (за исключением гл. 10, см. табл. П.1), в которой рассматривается пример, и порядкового номера примера в тексте главы.

Некоторые приложения используют в качестве источника данных демонстрационную базу данных Delphi, которая при стандартной инсталляции находится в папке ..\Program Files\Common Files\Borland Shared\Data. При работе с такими приложениями вам необходимо самостоятельно настроить соответствующие свойства компонентов доступа к данным. Они преднамеренно обнулены, так как на вашем компьютере база данных Delphi может располагаться в другом месте.

Ниже приводится описание демонстрационных приложений.

Таблица П.1. *Описание демонстрационных приложений*

Папка	Приложение
01_1	Группа проектов. Проект серверного приложения реализует сервер COM в составе внутреннего сервера (динамическая библиотека), который предоставляет два интерфейса с арифметическими функциями. Проект клиентского приложения демонстрирует использование этих интерфейсов
02_1	Группа проектов. Проект серверного приложения реализует простейший сервер Автоматизации. Клиентское приложение вызывает методы интерфейса сервера Автоматизации
03_1	Пример создания собственного элемента управления AciveX

Таблица П.1 *(продолжение)*

Папка	Приложение
04_1	Группа проектов. Проект сервера MTS на основе модуля данных MTS содержит простой интерфейс. Компонент `TADOConnection` необходимо настроить на демонстрационную базу данных. Клиентское приложение использует этот интерфейс. Для компонента `TDCOMConnection` необходимо задать имя компьютера-сервера
04_2	Группа проектов. Два проекта серверов MTS обеспечивают получение информации из демонстрационной базы данных. Компоненты `TADOConnection` необходимо настроить на демонстрационную базу данных. Дополнительный проект реализует объект MTS для распределенной транзакции. Компонент `TDCOMConnection` необходимо настроить на компьютер объекта MTS и при необходимости задать сам объект. Клиентское приложение получает данные от двух серверов MTS, используя распределенную транзакцию
06_1	Приложение показывает варианты использования компонентов dbExpress. В качестве источника данных выбран сервер Interbase, и для доступа к нему используется стандартный псевдоним BDE IBLocal. В случае отсутствия этого псевдонима необходимо задать вручную путь к файлу DBDEMOS.GDB демонстрационной базы данных для Interbase в компоненте `TSQLConnection`
07_1	Пример использования компонентов ADO. Путь к источнику данных задается пользователем во время выполнения приложения
09_1	Группа проектов. Пример распределенного приложения на основе технологии DataSnap. Серверное приложение для доступа к данным использует компоненты ADO и разделено на два модуля данных – главный и вторичный. Путь к базе данных хранится в файле INI. Описание *см. в гл. 9.* Клиентское приложение использует компоненты `TClientDataSet`. Описание *см. в гл. 10*
11_1	Пара приложений, использующих сокеты для пересылки данных
13_1	Пример использования CryptoAPI – менеджер установленных в системе сертификатов
14_1	Пример загрузки XML-файла в виде древовидной структуры в компонент TtreeView
14_2	Загрузка/выгрузка и просмотр ADO Recordset в виде XML
14_3	Расширенный пример 14_1 с возможностью редактирования
15_1	Пример преобразования данных XML в пакет данных Delphi и использования их как документа XML и набора данных

Таблица П.1 *(окончание)*

Папка	Приложение
16_1	Пример простого серверного Web-приложения, использующего демонстрационную базу данных Delphi. Перед началом работы необходимо настроить компонент `TADOTable`. В качестве сервера используется утилита Web App Debugger
16_2	Пример простого серверного Web-приложения, использующего демонстрационную базу данных Delphi. Перед началом работы необходимо настроить компонент `TADOTable`. Web-сервер – MS IIS. Динамическая библиотека помещается в папку Scripts сервера (это настройка по умолчанию) и вызывается из Web-браузера так: **http://localhost/scripts/simplewebapp.dll**
17_1	Простой пример клиента WEB-службы BASE, иллюстрирующий два способа использования компонента `THTTPRIO` и применение нестандартных типов данных
19_1	Web-приложение на основе технологии WebSnap и шаблонов HTML. Позволяет просматривать файлы и обеспечивает доступ к базам данных для просмотра и редактирования. Используется аутентификация пользователей. В качестве сервера применяется утилита Web App Debugger
19_2	Web-приложение на основе технологии WebSnap и шаблонов XML. Позволяет просматривать файлы и обеспечивает доступ к базам данных для просмотра и редактирования. В качестве сервера используется утилита Web App Debugger

Предметный указатель

A

ActiveX 2
ADO 221

C

CLSID (Class Identifier) 15
CoClass 18
CoCreateInstance 16
CoGetClassObject 17
COM (Component Object Model) 7
ConnectionString 230
Cookies 527
Critical section 415

D

DataSnap 313 318
dbExpress 187
Dispinterface 55
DTD (Document Type Definition) 429

E

EReconcileError 334
Event 415

G

GUID (Globally Unique IDentifier) 13, 22

I

IAccessor 226
IAppServer 114, 293, 295, 314
IClassFactory 17, 23
IClassFactory2 23
IColumnsInfo 226
ICommand 227
ICommandPrepare 228
ICommandProperties 228
ICommandText 228
ICommandWithParameters 228
IConnectionPointContainer 59
IConvertType 226
IDBCreateSession 225
IDBInitialize 225
IDBProperties 225
IDispatch 50
IDL (Interface Definition Language) 11
IDL CORBA 30
IDL Microsoft 30
IID (Interface Identifier) 13
IInterface 21
Invoke 50, 63
IObjectContext 104, 106, 109
IObjectControl 108, 109
IOpenRowset 226
IProvideClassInfo 21
IProviderSupport 306, 316
IRowset 226

IRowsetChange 227
IRowsetIdentity 227
IRowsetIndex 227
IRowsetInfo 226
IRowsetLocate 227
IRowsetUpdate 227
ISAPI-расширение 506
ISecurityProperty 106
ISourcesRowset 224
ISQLCommand 215
ISQLConnection 214
ISQLCursor 216
ISQLDriver 213
ISupportErrorInfo 66
ITransaction 226
ITransactionContext 103
ITransactionJoin 226
ITransactionLocal 226
ITransactionObject 226
ITypeInfo 18
ITypeLib 18
IUnknown 14, 19, 23

M

Microsoft ActiveX Data Objects 221
Mutex 415

O

OLE DB 222

P

Parser 429

R

Remote Interfaced Object (RIO) 563
RPC (Remote Procedure Call) 15
Runtime Type Information, RTTI 569

S

Semaphore 415

T

TADOCommand 258
TADOConnection 230
TADODataSet 256
TADOQuery 258
TADOStoredProc 258
TADOTable 257
TAutoIntfObject 66
TAutoObject 57
TAutoObjectFactory 65
TBDEClientDataSet 316
TClientDataSet 314, 317, 491
TComClassManager 26
TComObject 18, 19, 67
TComObjectFactory 23
TComServer 27
TConnectionBroker 288
TCORBAConnection 284
TCORBADataModule 296
TCustomADODataSet 244
TCustomClientDataSet 316
TCustomSQLDataSet 199
TCustomWebDispatcher 513
TDataSet 141, 153, 317
TDataSetField 331
TDatasetPageProducer 529
TDataSetProvider 292, 302, 316
TDatasetTableProducer 530
TDataSource 134, 138, 161
TDCOMConnection 279
TField 174
TGUID 22
TIBClientDataSet 316
TIndexDef 157
TIndexDefs 156
TInterfacedObject 22
TLocalConnection 287
TMTSAutoObject 109
TMTSDataModule 114, 296
TParam 160
TParameter 255
TParameters 253, 254
TQueryTableProducer 530

TRemoteDataModule 296
TSharedConnection 288
TSimpleObjectBroker 286
TSOAPDataModule 296
TSocketConnection 280
TSQLClientDataSet 205, 316
TSQLConnection 190, 195, 197
TSQLDataSet 201
TSQLMonitor 217
TSQLQuery 203
TSQLQueryTableProducer 530
TSQLStoredProc 204
TSQLTable 202
TThread 404
TTypedComObject 21
TTypedComObjectFactory 23, 26
TWebActionItem 516
TWebApplication 511
TWebConnection 283
TWebDataModule 296
TWebDispatcher 510
TWebModule 513
TWebRequest 517
TWebResponse 517
TXMLBroker 493
TXMLTransform 486
TXMLTransformClient 491
TXMLTransformProvider 489

W

Web-запрос 517
Web-модуль 513
Web-ответ 517
Web-сервер 502
wrapper 12

X

XML (eXtensible Markup Language) 427
XML-анализатор 429
XML-схема 429

А

Автоматизация 50
Агрегатные поля 329
Агрегаты 326
Агрегаты-объекты 327
Активизация just-in-time 106
Асимметричное шифрование 369
Аутентификация 370

Б

Библиотека COM 10, 15
Библиотека типов 11, 18, 28, 38
Буфер Delta 320

В

Взаимное исключение 415
Виды полей 175, 178
Вложенные наборы данных 331
Внешний провайдер данных 316
Внутренний провайдер данных 316
Входящий интерфейс 6
Вызов удаленных процедур 15
Вычисляемое поле 181

Г

Глобальный объект приложения 509
Глобальный уникальный идентификатор 13
Гонки 413
Группа с разделяемыми свойствами 116
Группировка агрегатов 330

Д

Действие 510, 516
Действие MTS 121

Дейтаграмма 350
Декларативная защита данных 105
Динамические поля 172
Диспетчерский идентификатор 51
Диспинтерфейс 51
Драйверы dbExpress 190

З

Закладка 169
Зашифрованная последовательность данных (ciphertext) 368

И

Идентификатор интерфейса 13
Идентификатор класса 15
Индекс 155
Интерфейс 8, 12
Интерфейс IDispatch 54
Интерфейс диспетчеризации 55
Интерфейс диспетчерский 51
Интерфейс дуальный 52, 56
Исходный текст (plaintext) 368

К

Клиент многозвенных приложений 313
Команда ADO 227
Компонент-адаптер 598
Компонент-диспетчер 597
Компонент-продюсер 511, 518, 598
Компоненты отображения данных 140
Контейнер элементов управления 5
Контекст объекта MTS 104
Контроллер Автоматизации 53, 67
Корневой элемент 434
Критическая секция 415

М

Маршалинг 10
Маршалинг диспинтерфейсов 53
Модель взаимодействия открытых систем (OSI) 344
Модуль приложения 600
Модуль страниц HTML 600
Моникер 11, 17

Н

Набор данных 133, 141
Номер порта 354

О

Объект COM 8, 11, 18, 31
Объект Автоматизации 53, 56
Объект поля 171
Объект типа stateful 102
Объект типа stateless 102
Объект транзакции MTS 101
Объект-источник данных ADO 225
Объект-команда ADO 227
Объект-набор рядов ADO 226
Объект-перечислитель ADO 224
Объект-приемник 59
Объект-сессия ADO 225
Объект-транзакция ADO 226
Объекты синхронизации 414
Отношение многие-ко-многим 166
Отношение один-ко-многим 165

П

Пакет MTS 102
Пакет данных 319, 479
Параметры запроса SQL 159
Первичный индекс 158, 323
ПО промежуточного слоя 273
Подписанный хэш 372
Позднее связывание 50
Поле типа BLOB 143
Пользовательский интерфейс 133
Поставщик криптографических услуг 379
Потоки 397

Провайдер ADO 223, 228
Провайдер данных 302
Программная защита данных 106
Пространство имен 441
Публикация 501
Пулинг объектов 108
Пулинг ресурсов 107

Р

Раннее связывание 50
Распределители ресурсов MTS 115
Редактор библиотеки типов 29
Роль MTS 105

С

Семафор 415
Сервер COM 9, 14
Сервер Автоматизации 53, 66
Сервер БД 273
Сервер приложения 291
Серверное Web-приложение 502
Сертификат 374
Сериализация 584
Симметричное шифрование 369
Событие 415
Стандарт DN 382
Статические поля 172
Страницы свойств 7

Т

Тег 427
Технология Web Broker 501
Типы данных 184
"Тонкие" клиенты 273
Транзакция MTS 103
Трансформационный файл XML 478, 485
Трехзвенная модель 273
Тупики 413

У

Удаленный модуль данных 276, 295
Удаленный модуль данных MTS 114
Утилита XML Mapper 480, 481

Ф

Фабрика класса 11, 17, 23
Фабрика класса Автоматизации 65
Фильтр 167
Фоновые процедуры 400

Х

Хэш 371

Ц

Целостность данных 371