Андрей Боровский

C++ и Pascal в Kylix 3

Разработка интернет-приложений и СУБД

- Взаимодействие приложений Kylix 3 с операционной системой Linux
- Интернет-программирование
- Принципы разработки интернет-приложений в Kylix 3
- СУБД InterBase. СУБД MySQL
- Разработка и распространение профессиональных программных продуктов

МАСТЕР_ПРОГРАММ_

Андрей Боровский

C++ и Pascal в Kylix 3

Разработка интернет-приложений и СУБД

Санкт-Петербург
«БХВ-Петербург»
2003

УДК 681.3.06
ББК 32.973.26-018
Б83

Боровский А. Н.

Б83 С++ и Pascal в Kylix 3. Разработка интернет-приложений и СУБД. — СПб.: БХВ-Петербург, 2003. — 544 с.: ил.

ISBN 5-94157-281-6

Книга рассказывает о новейших технологиях программирования на языках C++ и Pascal, реализованных в среде Kylix 3, о поддержке XSL и интерактивной отладке Web-приложений в Kylix IDE. Подробно рассматриваются такие технологии, как WebSnap и WebServices. Описываются особенности низкоуровневого программирования графического интерфейса (взаимодействие с библиотекой Qt library). Изложены такие важные для Kylix-программиста вопросы, как настройка Web-сервисов, создание резидентных Linux-приложений (демонов) и методы решения специфических проблем, возникающих при распространении Kylix-приложений.

Для программистов, имеющих базовые навыки работы в средах Delphi, Borland C++ Builder и предыдущих версиях Kylix

УДК 681.3.06
ББК 32.973.26-018

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Ольга Гурьева*
Компьютерная верстка	*Татьяны Олоновой*
Корректор	*Вера Александрова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 18.03.03.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 43,86.
Тираж 3000 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 5-94157-281-6

Содержание

Предисловие

Книга, которую вы держите в руках, посвящена практическому использованию среды разработки Kylix 3. Под практическим использованием понимается создание полезных программ (а не только учебных образцов). Можно полагать также, что читатели, собирающиеся использовать среду Kylix для разработки реальных программ, уже знакомы с основами программирования вообще и программированием в визуальных средах в частности и нуждаются в более углубленном изучении возможностей Kylix. Именно на этих предположениях и основана концепция данной книги.

Книга включает в основном методы реализации различных программных технологий в Kylix-приложениях. Некоторые из рассмотренных в книге технологий широко распространены в мире программирования, как, например, технологии SOAP и XSL, другие являются скорее специфичными для продуктов Borland (например, использование XML-документов в качестве локальных баз данных).

Подобный подход определяет и структуру книги. Каждый тематический раздел, посвященный программированию определенного типа приложений в среде Borland Kylix, предваряется введением, рассказывающим об основных принципах соответствующей технологии.

Казалось бы, сама идея визуального программирования нацелена именно на то, чтобы сделать процесс программирования как можно более наглядным. Это относится не только к разработке пользовательского интерфейса приложения. Например, работа с компонентами, обеспечивающими связь с базами данных, организована в Kylix таким образом, что еще на этапе разработки можно *видеть*, как создаваемое приложение будет взаимодействовать с выбранной СУБД (и будет ли оно взаимодействовать с ней вообще). То же самое можно сказать и о технологии разработки Web-приложений WebSnap, которая позволяет просматривать динамические страницы, создаваемые приложением, не покидая интегрированной среды разработки. Ко всему этому следует добавить прекрасную интерактивную справочную систему и объемистые руководства разработчика, поставляемые вместе с Kylix как в бумажном, так и в электронном формате.

И тем не менее мы убеждены, что книги о Kylix необходимы. Сколь бы подробна ни была справочная система, она не может охватить всего. Посе-

тите любой сайт, посвященный программированию для Delphi/Kylix, и вы найдете десятки статей, дополняющих и разъясняющих фирменную документацию. Зайдите на один из множества форумов по Delphi/Kylix-программированию, и вы увидите вопросы, на которые фирменная документация не дает развернутых ответов.

При написании книги автор старался (насколько это возможно) не дублировать справочную систему. Поэтому книга не содержит подробного перечисления и описания всех компонентов Kylix (учитывая общее их количество, это было бы не только нецелесообразно, но и трудноосуществимо). В соответствии с концепцией книги и предполагаемым уровнем подготовки читателя в ней раскрыты основные принципы реализации различных современных программных технологий в Borland Kylix 3, а также принципы разработки приложений, использующих эти технологии (в связи с чем, книга содержит немало практических пошаговых инструкций и примеров текстов готовых программ). Освоив эти принципы, читатель без труда сможет ориентироваться в соответствующих технологиях, прибегая, где это нужно, к услугам встроенной справочной системы.

Кроме перечисленного выше в книге затронута и специфика программирования в ОС Linux, выходящая за пределы Kylix-программирования, как, например, особенности файловой системы и межпроцессного взаимодействия в ОС Linux, настройка Web-серверов и серверов баз данных для работы с Kylix-приложениями, создание дистрибутивов приложений для ОС Linux и т. п.

В общем автор стремился максимально выйти за рамки тех сведений, которые можно получить с помощью клавиши <F1>, а потому книга окажется полезной для широкого круга программистов средней и высокой квалификации.

Введение

Эта книга, как следует из ее названия, посвящена программированию в среде разработки Kylix 3 на языках C++ и Delphi Language (так теперь называется версия Object Pascal).

Что такое Kylix 3?

Borland Kylix 3 представляет собой интегрированную среду разработки, предназначенную для быстрой визуальной разработки приложений на языках C++ и Delphi Language, способных выполняться отдельно от среды. При этом основной упор сделан на быстрой разработке графических интерфейсов (при помощи богатейшего набора графических компонентов, основанного на библиотеке Qt library), разработке Web-приложений и приложений для работы с базами данных. Разработчики Kylix не только стремятся к внедрению в свои продукты новейших технологий, но и делают все возможное для того, чтобы облегчить пользователям освоение новых методов программирования. Для ускорения и упрощения процесса разработки приложений в среде Borland Kylix введен целый ряд специальных функций и возможностей, начиная с первоклассного редактора графического интерфейса и "интеллектуального" редактора исходного текста, позволяющего ускорить ввод текста программы, и заканчивая такими средствами, как интерактивный отладчик приложений для Web-сервера или модель визуальной разработки приложений баз данных. В рамках этой модели взаимодействие с сервером баз данных может выполняться еще на этапе разработки приложения, что делает этот процесс более наглядным и позволяет обнаруживать и устранять на данном этапе некоторые ошибки, проявляющиеся обычно только во время выполнения программы.

Среди достоинств последних версий Kylix нельзя не отметить появление средств разработки, отражающих передовые тенденции в сетевом программировании, в частности, широкого спектра инструментов, которые предназначены для работы с XML-документами и Web-службами. Уникальные решения, предоставляемые Borland, позволяют интегрировать поддержку технологий XML и XSL в приложения самых разных типов, в том числе в локальные и распределенные приложения баз данных.

Появление в новой версии Kylix интегрированной среды разработки на языке C++ позволяет не только использовать мощь и гибкость этого языка, но и напрямую обращаться к различным интерфейсам программирования ОС Linux.

Ко всему перечисленному выше следует добавить подробную и удобную в использовании интерактивную справочную систему, которая позволяет получить доступ не только к справке по функциям Kylix и элементам реализованных в этой среде языков программирования, но и к встроенной в операционной системе Linux справочной системе man.

Средства разработки Kylix совместимы с аналогичными продуктами Borland для платформы Win32, что делает возможным перенос проектов между платформами ОС Linux и Windows. Вместе с тем Borland Kylix обладает рядом существенных отличий, связанных прежде всего со спецификой ОС Linux и средствами взаимодействия с Интернетом и межпроцессного взаимодействия, реализованными в этой операционной системе. Неизбежные различия возникают в связи с принципиальной разницей интерфейсов прикладного программирования (API) ОС Linux и Windows, а также вследствие некоторых особенностей, присущих вариантам языков C и C++, традиционно используемых в ОС Linux.

Длительная история разработки компанией Borland визуальных сред программирования для ОС Windows и Linux позволяет сделать однозначные выводы о стратегических направлениях развития этих средств. Очевидно, что основной задачей Borland является создание средств быстрой разработки приложений для корпоративного сектора, а также приложений для ведения электронного бизнеса в Глобальной сети.

По сравнению с предыдущими версиями в Kylix 3 введен целый ряд дополнений. О главном новшестве — компиляторе языка C++ уже говорилось. Среди других достоинств версии 3 заслуживают внимания следующие:

- поддержка инструкций Pentium 4 во встроенном ассемблере;
- предварительная компиляция заголовочных файлов C++ (precompiled headers) для ускорения компиляции приложений;
- встроенный интерактивный отладчик Web-приложений;
- редактор связей между объектами (окно **Diagram**) позволяет отображать и редактировать эти связи через свойства;
- поддержка работы с XSL-шаблонами (компонент XSLPageProducer);
- новые драйверы набора компонентов DBExpress.

Зачем нужен Kylix 3?1

Подобный вопрос приходится слышать нередко, причем те, кто его задает, иногда напоминают, что для ОС Linux уже существуют бесплатные реализации языка Pascal. К ним относится интегрированная среда Lazarus, стремящаяся, как говорят ее разработчики, стать максимально похожей и полностью

совместимой, по крайней мере, для исходных текстов, со средой Delphi (но пока этот проект весьма далек от намеченной цели). Те, кто задают подобные вопросы, забывают, что Kylix — это не просто "Object Pascal для Linux", что, кстати говоря, вообще перестало быть актуальным с введением в Kylix 3 среды программирования C++. Borland Kylix — это прежде всего средство разработки наиболее распространенных на сегодняшний день типов приложений, использующих последние технологии в таких областях, как интернет-программирование и работа с базами данных. Осмелимся утверждать, что сегодня в мире ОС Linux нет другого средства разработки, позволяющего решать те же задачи с той же быстротой и удобством. Если до недавнего времени программистов, работающих под ОС Linux, могла отпугивать необходимость использовать при работе с Kylix нелюбимый многими язык Pascal, то теперь с появлением интегрированного пакета, содержащего средства разработки как на языке Delphi Language, так и на языке C++, эта проблема также отпадает. Разумеется, Borland Kylix не претендует (и никогда не претендовал) на роль средства системного программирования. Этот продукт предназначен для быстрой разработки прикладных программ. Если Kylix Open Edition предоставляет в распоряжение пользователя удобное средство визуальной разработки и широкий набор графических компонентов (не говоря уже о компиляторах языков Delphi Language и C++ и удобнейшем редакторе исходного текста), то Kylix Enterprise вполне может служить основой для разработки корпоративных решений в рамках предприятия.

Что вы найдете в этой книге?

Эта книга посвящена практическому использованию различных технологий программирования, реализованных в Kylix 3. При этом основной акцент сделан на новых компонентах и технологиях, появившихся в версиях 2 и 3 Kylix. Перед тем, как описать содержимое книги, будет весьма уместно перечислить то, чего читатель в ней не найдет.

В книге не излагаются основы языков программирования Delphi Language и C++, а также концепции объектно-ориентированного программирования и работы в интегрированной среде разработки. Отсутствует описание простейших методов визуального программирования и построения стандартных графических интерфейсов приложений. Если читатель нуждается в подобных сведениях, можно рекомендовать предварительно прочесть одну из книг, являющихся введением в программирование для Delphi/Kylix. Поскольку многие из тех, кто выбирает Kylix в качестве средства разработки, уже знакомы с этими вводными вопросами (например, по опыту программирования в среде Delphi), представляется предпочтительным сосредоточить внимание на более сложных (и не столь широко освещенных) аспектах программирования в Kylix для ОС Linux.

Три первые части книги соответствуют трем основным сферам применения Borland Kylix. *Часть I* представляет собой введение в прикладное программирование для ОС Linux. Эта часть предназначена для разработчиков, незнакомых с основами программирования в указанной операционной системе, а также для Linux-программистов, стремящихся "адаптироваться" к среде программирования Borland Kylix. *Глава 1* данной части включает сведения о разработке консольных приложений и разделяемых библиотек ОС Linux с помощью Borland Kylix, взаимодействии Kylix-приложений с интерфейсами прикладного программирования ОС Linux, особенностях работы с файловой системой этой ОС (включая виртуальные файловые системы), механизмах управления процессами и межпроцессного взаимодействия. *Глава 2 части I* посвящена тонкостям программирования графического интерфейса Kylix-приложений (рассматриваются примеры на языках Delphi Language и C++) и описанию взаимодействия графических компонентов Kylix с графической подсистемой ОС Linux. В этой главе читатель найдет также примеры методов запуска графических приложений из Kylix-приложений, а также использования в Kylix-приложениях альтернативных графических интерфейсов.

Изложение вышеуказанного материала сопровождается необходимыми пояснениями, касающимися принципов работы ОС Linux и, естественно, примерами, написанными с использованием как языка Delphi Language, так и языка C++. При этом определенное внимание уделено особенностям реализации языка C++ в Kylix и соотношению этих особенностей с традициями программирования на языках C и C++, принятыми в ОС Linux.

Часть II книги посвящена интернет-программированию. Поскольку часть II посвящена разработке приложений для Web-серверов, в первой главе этой части *(глава 3)* дается краткое описание сервера Apache и других Web-серверов, распространенных на платформе ОС Linux, излагаются принципы разработки модулей для сервера Apache и CGI-приложений. *Глава 4* является вводно-обзорной главой, кратко описывающей методы разработки сетевых приложений, более подробно изложенных в последующих главах. В этой главе приводятся примеры создания простейших интернет-приложений "с нуля" без использования специфических компонентов Kylix. Основная цель этих примеров заключается в том, чтобы помочь читателю освоить принципы работы сетевой подсистемы ОС Linux (хотя в определенных ситуациях эти примеры, демонстрирующие возможности разработки "легких" сетевых приложений с помощью Kylix, могут быть полезны и сами по себе). Далее следует краткое описание специальных средств разработки сетевых приложений различных типов, а также средств отладки этих приложений.

Глава 5 посвящена разработке независимых интернет-приложений на основе набора компонентов Internet Direct (Indy). При этом подробно рассматривается многопоточная модель Indy, принципы реализации TCP и UDP-

протоколов с помощью этого набора компонентов, приводятся примеры приложений, осуществляющих с помощью Indy работу с популярными интернет-протоколами HTTP и FTP.

Поскольку программирование интернет-приложений в Kylix тесно связано с новейшими интернет-технологиями, автор счел необходимым посвятить отдельную *главу 6* описанию языка XML и его производных. Также в этой главе рассматриваются инструменты Borland Kylix, предназначенные для работы с языком XML.

Три следующие главы 7, 8 и 9 части III посвящены "трем китам" Web-программирования в Borland Kylix — технологиям WebBroker (быстрая разработка Web-приложений и динамическая генерация контента), WebSnap (автоматизация распространенных задач Web-программирования и технология сценариев на стороне сервера) и WebServices (построение распределенных приложений на основе протокола SOAP). В рамках каждой главы дается описание основных компонентов Kylix, реализующих соответствующую технологию, и рассматриваются примеры использования этих компонентов в приложениях для Web-сервера, написанных на языках C++ и Delphi Language.

При описании технологии WebBroker подробно рассматривается механизм диспетчеризации запросов, формирования HTTP-ответов и использования компонентов-генераторов контента. В качестве примера приводится Web-приложение, использующее технологию cookies для идентификации пользователей.

В главе 8, посвященной технологии WebSnap, приводятся примеры написания сценариев на стороне сервера на языке JavaScript, описываются механизмы работы компонентов-адаптеров, команд и полей компонентов-адаптеров различных типов. Эта глава включает также примеры использования других важных компонентов технологии WebSnap и приемов визуального программирования Web-приложений в рамках данной технологии.

Глава 9, посвященная технологии WebServices, начинается с краткого описания протокола SOAP и языка WSDL. Объяснение принципов работы компонентов WebServices и других инструментов Kylix, предназначенных для работы с этой технологией, сопровождается примерами распределенных SOAP-систем, написанных на языках C++ и Delphi Language.

Завершает *часть II* книги *глава 10*, посвященная описанию программирования распределенных приложений на основе технологии CORBA с использованием брокера запросов VisiBroker.

Часть III книги посвящена разработке локальных и распределенных приложений баз данных с помощью Borland Kylix. При этом, как и в предыдущих частях, примеры предваряются необходимыми теоретическими сведениями.

В главе 11 излагаются основные принципы реляционной модели баз данных и дается описание общей архитектуры приложений баз данных в Borland Kylix.

Главы 12 и *13* посвящены описанию настройки и работы с двумя распространенными в мире ОС Linux системами управления базами данных (СУБД) — InterBase и MySQL.

Поскольку компоненты dbExpress, являющиеся основой механизма взаимодействия Kylix-приложений с базами данных, используют язык запросов SQL, описание компонентов dbExpress, которому посвящена *глава 14*, предваряется кратким описанием языка SQL.

В главе 15, посвященной локальным приложениям баз данных, рассматривается работа с клиентскими наборами данных, компонентами-провайдерами и элементами пользовательского интерфейса приложений баз данных. На примерах нескольких баз данных описываются такие операции, как поиск и фильтрация записей, создание индексов и закладок, хранене в базах данных изображений и т. п. Также в этой главе указаны средства и методы хранения данных в XML-документах и механизмы взаимодействия между приложениями, использующими в качестве хранилища данных XML-документы и классические приложения баз данных.

Глава 16, посвященная описанию распределенных приложений баз данных, начинается с модели "портфеля", которая обеспечивает возможность работы клиентских программ распределенных систем баз данных в автономном режиме. Далее следует представление многоуровневой модели распределенных приложений баз данных, реализованной на основе протокола SOAP. Заключает главу 16 описание методов разработки Web-приложений баз данных, в рамках которого рассматривается применение в приложениях баз данных технологий WebBroker и WebSnap.

Заключительная, *часть IV* книги содержит дополнительные главы, посвященные проблемам, возникающим перед разработчиками готовых программных продуктов.

В главе 17 рассматриваются вопросы разработки компонентов для среды Borland Kylix. Излишне напоминать, что в настоящее время компоненты для интегрированных сред разработки часто распространяются как коммерческие программные продукты.

Глава 18 посвящена вопросам сопровождения и распространения Kylix-приложений. В этой главе читатель найдет разделы, посвященные созданию справочной системы для приложений Kylix, а также различным методам распространения готовых приложений.

Глава 19 — единственная глава книги, непосредственно не связанная с программированием в Borland Kylix. В этой главе рассматриваются особенности различных архитектур приложений электронного бизнеса, на которые ориентировались разработчики Kylix.

Для кого предназначена эта книга?

С учетом уровня изложения материала и концепции книги от читателя требуется наличие определенных предварительных знаний. Примеры программ в книге приводятся на языках программирования C++ и Delphi Language, так что предполагается владение хотя бы одним из этих языков. Предполагаются также знакомство с концепциями объектно-ориентированного программирования и навыки разработки приложений в визуальной среде.

Таким образом, можно представить себе потенциального читателя книги как программиста, имеющего опыт работы с Delphi или C++ Builder и желающего освоить разработку приложений на платформе ОС Linux, либо как Linux-разработчика, стремящегося освоить технологии программирования интернет-приложений и приложений баз данных, предоставляемые средой Borland Kylix.

Впрочем, можно надеяться, что книга окажется полезной для всех, кто интересуется программированием в средах Borland или программированием для ОС Linux.

Часть I

Взаимодействие приложений Kylix 3 с операционной системой

Глава 1

Kylix-приложения и прикладные интерфейсы Linux

В этой главе мы рассмотрим основные аспекты взаимодействия приложений, созданных в среде Kylix 3, с ОС Linux. Данная глава посвящена особенностям разработки консольных приложений, работе с файловой системой Linux, взаимодействию приложений, написанных на языке C++ и Delphi Language, с системными библиотеками, механизмам взаимодействия между процессами и особенностям программирования на языках C/C++ в ОС Linux.

Особенности языка C++ в Borland Kylix

Особенности реализации языка C++ в среде разработки Borland Kylix 3 определяются, с одной стороны, традициями компании Borland, а с другой стороны, спецификой ОС Linux. Если у вас есть опыт работы с Borland C++ Builder, новая среда для Linux не должна показаться вам незнакомой, ведь основной задачей Borland и было создание среды разработки, облегчающей перенос C++ проектов из среды Windows на платформу Linux. Тем не менее, некоторые особенности ОС Linux потребовали от разработчиков введения новых возможностей. Далее мы рассмотрим некоторые характерные черты Kylix C++, отличающие этот язык от варианта C++ Builder и от знакомого Linux-программистам GNU C++.

Первая характерная особенность Borland C++ Builder связана с директивами компилятора. В этом варианте традиционно используется ряд специфических директив `#pragma`. Собственно говоря, смысл директив `#pragma` и заключается в том, что эти директивы позволяют компилятору устанавливать опции, не конфликтующие с опциями других компиляторов. Если компилятор встречает в тексте программы незнакомую директиву `#pragma`, он ее просто игнорирует. Использование директив `#pragma` позволяет перенести в текст программы многие опции компилятора Kylix, обычно сохраняемые в специальных файлах проектов. Таким образом, с помощью

директив `#pragma` можно создавать программы, предназначенные для компиляции разными компиляторами C++.

Полный список директив `#pragma` приводится в справочной системе Kylix C++ IDE. Мы же в качестве примера рассмотрим здесь лишь две директивы — `#pragma link` и `#pragma hdrstop`. Директива `#pragma link` позволяет указывать список модулей, связываемых с программой на этапе компоновки, например, для того чтобы связать приложение с библиотекой libGL.so указываем:

```
#pragma link "/usr/X11R6/lib/libGL.so"
```

Как видим, `#pragma link` позволяет указать не только имя модуля, но и путь к нему, чем эта опция выгодно отличается от директив связывания разделяемых библиотек в Delphi Language IDE. С помощью директивы `#pragma link` можно связывать с приложением и объектные файлы:

`#pragma link` "objfile.o" (в Linux объектные файлы имеют расширение .O).

Впрочем, в C++ IDE связывать внешние модули с проектом можно при помощи команд меню **Project | Add to Project**.

Директива `#pragma hdrstop` находится в тесной связи с другой особенностью Borland C++ — предварительной компиляцией заголовков. Предварительная компиляция заголовков предназначена для того, чтобы хотя бы отчасти преодолеть один из основных недостатков компиляторов C++ — низкую скорость сборки приложений.

Существенно более низкая скорость компиляции проектов на C++ по сравнению с программами, написанными на стандартном C, связана отчасти с гораздо более сложным анализом заголовочных файлов, содержащих объявления классов C++, который приходится выполнять компилятору. Для решения этой проблемы разработчики Borland и предложили использовать предварительную компиляцию заголовочных файлов (под компиляцией в данном случае подразумевается построение таблиц символов (symbol tables)). Выигрыш в скорости основан на том, что заголовочные файлы, как правило, — самая консервативная часть проекта, а заголовки стандартных библиотек разработчиками программ вообще не модифицируются. Таким образом, выполнив компиляцию заголовков один раз, можно существенно ускорить процесс сборки приложений.

Практически эта технология реализована следующим образом. Для каждого набора директив `#include`, встречающегося в исходных файлах проекта, создается отображение (image), содержащее скомпилированные заголовки в той последовательности, в которой они включены в исходный файл. Все отображения, созданные для данного проекта, сохраняются в специальном файле. Совместное использование одного отображения скомпилированных заголовков несколькими модулями возможно только в том случае, если заголовки включены в исходные файлы в одной и той же последовательности.

Поскольку секции исходных файлов, содержащие директивы `#include`, как правило, совпадают лишь частично, для совместного использования отображений скомпилированных заголовков и введена директива `#pragma hdrstop`. Если указанная директива встречается в исходном файле, предварительная компиляция выполняется только для заголовочных файлов, включенных перед этой директивой. Благодаря директиве `#pragma hdrstop` исходные файлы

```
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#pragma hdrstop
#include "unit1.h"
...
```

и

```
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#pragma hdrstop
#include "unit2.h"
...
```

могут использовать одно и то же отображение заголовков.

Включение/отключение режима предварительной компиляции заголовков управляется опциями раздела **Precompiled Headers** вкладки **Compiler** окна **Project Options** (по умолчанию режим включен). На этой вкладке можно также указать имя файла, в котором будут храниться скомпилированные заголовки для данного проекта (в противном случае компилятор сам выберет это имя). На этой же вкладке можно установить и значение **Stop After**, имеющее тот же смысл, что и директива `#pragma hdrstop`.

При желании вы можете включить в число предварительно компилируемых заголовков и те заголовочные файлы, что были созданы в рамках вашего проекта, если, конечно, вы уверены в их неизменности. Не следует включать в число предварительно компилируемых заголовочных файлов те, на которые ссылаются другие предварительно компилируемые заголовки.

Еще одна особенность Borland C++ связана с объектными файлами. В отличие от Delphi IDE, компилятор C++ генерирует объектные файлы для всех модулей проекта. Эти файлы полностью соответствуют стандарту объектных файлов компилятора GCC и экспортируют таблицы имен, которые можно просматривать с помощью стандартной Linux-команды `nm`.

Листинг 1.1. Фрагмент вывода команды `nm` для объектного файла модуля Kylix C++

```
00000000 W @TForm1@$bctr$qqrp18Classes@TComponent
00000000 W @TForm1@$bdtr$qqrv
00000000 W @TForm1@Button1Click$qqrp14System@TObject
00000000 W @TForm1@Button2Click$qqrp14System@TObject
00000000 W @TForm1@FormClose$qqrp14System@TObjectr19Qforms@TCloseAction
```

Таким образом, существует принципиальная возможность разделения объектных файлов между компиляторами Kylix и GCC. Однако на этом пути возникают существенные ограничения. Дело в том, что форматы хранения классов в объектных файлах Kylix и GCC различаются. Классы Borland и GCC различаются не только форматом вызова методов (в Kylix это fastcall, а в GCC — cdecl), но и схемой изменения имен (names mangling). В листинге 1.1 были приведены примеры экспортируемых имен методов, созданных компилятором Kylix. Сравним их с именами методов, сгенерированных компилятором GCC (листинг 1.2).

Листинг 1.2. Фрагмент вывода команды `nm` для объектного файла, созданного с компилятором GCC

```
0000002c T _._7QtGLDoc
00000000 T __7QtGLDoc
000000a4 T isModified__C7QtGLDoc
00000088 T load__7QtGLDocRC7QString
00000068 T newDoc__7QtGLDoc
0000007c T saveAs__7QtGLDocRC7QString
```

Как говорится, разница очевидна. Механизмы выделения и высвобождения памяти для экземпляров классов в Kylix и GCC также различаются. Одним из следствий различия форматов классов является то, что в Borland Kylix вы не сможете непосредственно использовать библиотеки Linux (как статические, так и динамические), экспортирующие классы C++. Разделение внешних модулей между GCC и Kylix возможно только в том случае, если эти модули написаны на стандартном языке C. GCC по умолчанию генерирует объектный код, основанный на правилах C, для всех файлов с расширением .c, а в Kylix для создания C-модуля следует выбрать пункт **C File** страницы **New** диалогового окна **New Items**.

Для программистов, привыкших к GNU C++, следует отметить такую особенность Borland C++, как свойства (properties). Свойства инкапсулируют поля классов, для чего в традиционном C++ приходится создавать многочисленные пары процедура/функция. Borland C++ позволяет определять свойства самых разных типов, включая такие, как указатель на экземпляр класса или массив записей.

Как и GCC, компилятор Borland позволяет использовать директивы встраивания функций в программах, написанных на языке C. Только в случае C в Borland Kylix следует использовать директиву `__inline`.

Как и в Borland C++ Builder, в Kylix C++ существует несколько встроенных макросов, предназначенных для условной компиляции кода. Прежде всего, это макрос `__BCPLUSPLUS__`, означающий, что программа компилируется в среде Borland C++. Кроме того, компилятор поддерживает стандартный макрос `__cplusplus`. Если вы пишете программу на стандартном C, то для выделения фрагмента кода, предназначенного для компилятора Borland, можно использовать макрос `__BORLANDC__` (значение этого макроса указывает номер версии компилятора). Для выделения платформо-зависимых фрагментов кода можно использовать макросы `#ifdef WINDOWS` и `#ifdef __linux__`.

Как уже отмечалось, среда Kylix 3 позволяет создавать программы и на C++, и на Delphi Language, поэтому в данной книге исходные тексты программ приводятся на этих языках. Дублировать текст каждой программы "переводом" на другой язык программирования было бы нерационально, тем более что большинство читателей этой книги, надеемся, владеет и C++, и Pascal. Тем не менее специально для тех, кто незнаком с одним из этих языков, приводится несколько замечаний, призванных облегчить "перевод" листингов. В табл. 1.1 приведено сравнительное описание типов в C++ и Delphi Language.

Таблица 1.1. *Сравнение типов C++ и Delphi Language*

Тип C++	Тип Delphi Language	Описание типа	Размер, байт
		Целые типы	
`Char`	`ShortInt`, `Char`	Знаковый 8-битный или символ	1
`unsigned char`	`Byte`	Беззнаковый 8-битный	1
`short`	`SmallInt`	Знаковый 16-битный	2
`unsigned short`	`Word`	Беззнаковый 16-битный	2
`int`, `long`	`Integer`, `LongInt`	Знаковый 32-битный	4
`unsigned long`, `unsigned int`	`LongWord`, `Cardinal`	Беззнаковый 32-битный	4
`long long`	`Int64`	Знаковый 64-битный	8
`unsigned long long`	—	Беззнаковый 64-битный	8

Таблица 1.1 *(окончание)*

Тип C++	Тип Delphi Language	Описание типа	Размер, байт
	Типы чисел с плавающей точкой		
`float`	Single	7—8 значащих цифр	4
`double`	`Double`	15—16 значащих цифр	8
`long double`	`Extended`	19—20 значащих цифр	10
	Логические типы		
bool	Boolean	Логический тип (`True`/`False`)	1
	Типы-указатели		
`void *`	`Pointer`	Нетипизированный указатель	4
`char *`	`PChar`	Указатель на тип `char`, массив типа `char`, строка, заканчивающаяся нулем	4, длина строки произвольная
`some_type *`	`^Some_Type`	Указатель на переменную типа `Type`	4

Отметим еще несколько особенностей C++ и Delphi Language. Для обращения к полям структуры (записи), заданной указателем, в C++ используется оператор `->`. Следующие конструкции эквивалентны:

C++:

```
some_struct->some_field
```

Стандартный Pascal:

```
some_struct^.some_field
```

Delphi Language:

```
some_struct.some_field
```

Программистам, пишущим на языке Pascal, следует учитывать еще одну особенность языков C/C++. В этих языках типы `указатель на тип` и `массив типа` формально взаимозаменяемы. Что имеется в виду в данном конкретном случае, следует смотреть по контексту. Например, в объявлении

```
void some_func(int * retval);
```

переменная `retval` скорее всего является указателем на переменную типа `int`, и в Delphi Language эту декларацию можно перевести как

```
procedure some_func(var retval : Integer); cdecl;
```

а вот в объявлении

```
int ** ppint;
```

переменная `ppint` может интерпретироваться и как массив указателей на переменную типа `int`, и как указатель на массив переменных типа `int`, и как указатель на указатель на переменную типа `int`. Для того чтобы понять, какой именно смысл имеет переменная, придется разбираться в логике работы модуля, содержащего это объявление. Стоит отметить, что неверная интерпретация смысла указателей является, пожалуй, самым частым источником ошибок при программировании. Учтите также, что в C/C++ индексация массивов всегда начинается с нуля, так что объявление переменной

```
int a[ARRAY_SIZE];
```

следует переводить на Pascal как

```
a : array[0..ARRAY_SIZE-1] of Integer;
```

Еще одна особенность C/C++ заключается в том, что в операциях сравнения могут использоваться не только логические, но и целые типы. При этом логическому значению `False` соответствует целое значение 0, а другие значения целочисленной переменной интерпретируются как `True`. Так что C-конструкцию

```
if(non_boolean_val) ...
```

можно перевести на язык Pascal, как

```
if non_boolean_val <> 0 then ...
```

Консольные приложения на языках C++ и Delphi Language

В этом разделе мы познакомимся с особенностями разработки консольных приложений (приложения для текстового терминала) для ОС Linux, функциями библиотеки glibc и методами обработки ошибок, возникающих при вызове этих функций. Разработчики, переходящие с платформы Windows на платформу Linux, должны иметь в виду, что в ОС Linux консольные приложения играют большую роль.

В ОС Linux графическая подсистема отделена от остальной операционной системы, и в принципе эта 32-разрядная многозадачная ОС может использоваться вообще без графического интерфейса. У такого подхода есть свои преимущества: при решении ресурсоемких задач вся мощность процессора может быть брошена на выполнение полезной работы, а не затрачиваться на перерисовку окон. Кроме того, приложениям с текстовым интерфейсом присуща бóльшая компактность, стабильность и меньшая зависимость от внешних библиотек, которые могут быть (а могут и не быть) установлены в конкретной системе. Сам интерфейс командной строки ОС Linux предос-

тавляет пользователю гораздо большие возможности, нежели командная строка DOS/Windows, и многие задачи в ОС Linux гораздо удобнее решать именно из командной строки. Многие важные типы приложений Linux, например приложения-демоны или внешние утилиты, предназначенные для использования в сценариях оболочки, должны быть консольными. Вот почему, изучая программирование для ОС Linux, следует уделить особое внимание вопросам создания консольных приложений.

В этом разделе будет рассмотрено несколько простых консольных программ — для Kylix C++ и Kylix Delphi. Мы также коснемся некоторых проблем, которые возникают в связи с программированием консольных приложений, но имеют значение и при разработке приложений других типов. Так что, даже если не стоит задача писать консольные программы, все равно полезно будет прочесть этот раздел.

При создании проекта консольного приложения C/C++ среда Kylix выводит дополнительное диалоговое окно (рис. 1.1). В этом диалоговом окне можно указать язык консольного приложения (C или C++) и необходимость поддерживать потоки в приложении. Если в качестве языка приложения вы выбрали C++, то с помощью специального флажка вы можете указать также, будет ли приложение использовать классы CLX или нет.

Полезной опцией является компонент ввода **Specify Project Source**, с помощью которого в проект можно импортировать уже готовые тексты C/C++ приложений.

Рассмотрим простое консольное приложение, написанное в Borland Kylix на языке C++ (листинг 1.3).

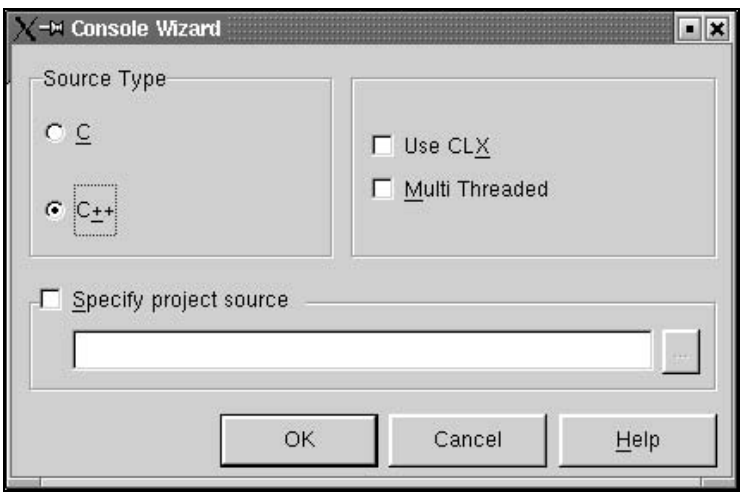


Рис. 1.1. Окно **Console Wizard**

Листинг 1.3. Консольное C++ приложение для Kylix 3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#pragma hdrstop

#pragma argsused

enum {ERROR = -1, SUCCESS, FAIL};

#define BUFSIZE 0x1000
static char buf[2][BUFSIZE];

int fcompare(const char *fname1, const char *fname2)
{
  FILE *f1, *f2;
  int retval = SUCCESS;
  if ((f1 = fopen(fname1, "rb")) == NULL)
  return ERROR;
  if ((f2 = fopen(fname2, "rb")) != NULL)
  {
    size_t size1, size2;
    do
    {
       size1 = fread(buf[0], 1, BUFSIZE, f1);
       size2 = fread(buf[1], 1, BUFSIZE, f2);
       if (0 == (size1 | size2)) break;
       if ((size1 != size2) || memcmp(buf[0], buf[1], size1))
       {
          retval = FAIL;
          break;
       }
    } while (size1 && size2);
    fclose(f2);
  }
  else  retval = ERROR;
```

```
  fclose(f1);
  return retval;
}

int main(int argc, char *argv[])
{
  if (argc != 3)
  {
    puts("Использование: %s file1 file2\n", argv[0]);
    return 0;
  }
  int result = fcompare(argv[1], argv[2]);
  printf("Файлы %s и %s %s\n", argv[1], argv[2],
           result == SUCCESS ? "совпадают" : "не совпадают");
  return result == SUCCESS ? 0 : 1;
}
```

Эта программа сравнивает два файла, переданных ей в качестве параметров при запуске. Программа не только выводит сообщение о совпадении или несовпадении файлов, но и возвращает коды выхода 0, если файлы совпали, и 1 — в противном случае, так что ее можно использовать и в командной строке, и в файлах сценариев.

Как видим, простейшая программа для Kylix C++ ничем не отличается от программ для других компиляторов. Для создания консольной программы на C/C++ в Kylix не нужно указывать специальных директив компилятора, как в случае с Delphi Language. Текст такой программы можно распространять без файла проекта (при открытии файла с программой в Kylix 3 C++ IDE файл проекта будет сгенерирован автоматически).

В случае программы, указанной выше, нам не следует беспокоиться о подключении системных библиотек, так как все функции, используемые в этой программе, определены в библиотеке `glibc`, которая автоматически подключается компоновщиком Kylix.

В случае если программе необходимы функции, экспортируемые другими библиотеками ОС Linux, эти библиотеки придется указывать явно либо в настройках проекта, либо при помощи директивы `#pragma link`.

Большая часть библиотек ОС Linux существует в двух вариантах: динамическом и статическом. Динамические библиотеки Linux подобны библиотекам DLL Windows. Эти библиотеки не включаются в код приложения и позволяют нескольким процессам разделять один и тот же файл. В ОС Linux динамические библиотеки хранятся в файлах с расширением .so. Отличительной особенностью динамических библиотек Linux является наличие у одной

библиотеки нескольких имен (подробнее этот вопрос будет рассмотрен в разделе, посвященном созданию разделяемых библиотек).

Статические библиотеки содержат объектный код, который статически связывается с программой во время ее компоновки. В ОС Linux файлы статических библиотек имеют расширение A. Фактически статические библиотеки можно рассматривать как совокупность объектных файлов. Компоновщики различных компиляторов находят эти библиотеки и включают их по мере необходимости в образ (image) исполнимой программы.

В отличие от Kylix Delphi, Kylix C++ позволяет использовать в программах не только динамические, но и статические библиотеки, так что перед разработчиком возникает вопрос: какой тип библиотек выбрать?

У каждого подхода есть свои достоинства и недостатки. Использование динамических библиотек позволяет уменьшить размер исполнимых модулей. Экономия особенно ощутима, когда несколько программ используют одну и ту же библиотеку. Специфическим неудобством динамических библиотек в ОС Linux (по сравнению с Windows) является то, что в разных дистрибутивах ОС Linux одни и те же библиотеки могут располагаться в разных каталогах и иметь разные имена. Последнее обстоятельство связано с тем, что в ОС Linux номер версии библиотеки является частью ее имени. Кроме того, многие библиотеки Linux не имеют обратной совместимости с прежними версиями. Для открытого программного обеспечения, распространяемого в виде исходных текстов и компилируемого специально для каждого дистрибутива, все это не представляет проблемы, а вот при распространении коммерческих приложений можно столкнуться с некоторыми трудностями. С одним из методов решения этой проблемы мы познакомимся в разделе "Разделяемые библиотеки и объектные файлы" этой главы.

При использовании статических библиотек размер файла программы может существенно увеличиться, но статические библиотеки являются частью программы и уменьшают ее зависимость от внешних модулей.

Говоря о библиотеке `glibc`, следует отметить особенность получения справочных данных по ее функциям. Справочная система Kylix не содержит сведений о функциях `glibc`. Вместо этого для получения справки о функции вызывается соответствующая страница встроенной справочной системы Linux man.

В начале раздела были перечислены некоторые причины, заставляющие программистов прибегать к консольному программированию.

Здесь уместно указать еще одну. В ОС Linux принципы управления процессами, в частности механизмы создания и удаления дочерних процессов, отличаются от соответствующих механизмов ОС Windows. Для того чтобы создать дочерний процесс, будь то копия основного процесса или новый

процесс, приложение ОС Linux должно сначала создать копию собственного процесса, а затем, если это необходимо, заместить созданный дочерний процесс другим процессом. В случае консольных приложений такие операции не вызывают затруднений, а вот с приложениями, обладающими графическим интерфейсом, могут возникнуть проблемы. В процессе взаимодействия графического приложения с графическим сервером сервер передает процессу различные уникальные идентификаторы, ссылающиеся на ресурсы, выделенные сервером специально для данного процесса. При копировании процессов дочерний процесс наследует все идентификаторы своего родителя. Попытка обращения двух процессов к одним и тем же ресурсам графического сервера может серьезно нарушить работу не только этих двух приложений, но и самого сервера. Конечно, это препятствие не является непреодолимым, и в следующей главе, посвященной графическим приложениям, мы рассмотрим методы управления дочерними процессами с учетом специфики этих приложений, однако в консольных приложениях запуск дочернего процесса выполняется гораздо проще.

В качестве примера приведем консольную С-программу, создающую дочерний процесс (листниг 1.4).

Листинг 1.4.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#pragma hdrstop

#pragma argsused

int main(int argc, char *argv[])
{
  pid_t pid;
  if (argc != 2)
  {
    puts("Использование: %s команда\n", argv[0]);
    return 0;
  }
  pid = fork();
  switch(pid)
```

```
  {
    case -1 :
       perror("Не удалось создать дочерний процесс");
       break;
    case 0 :
      printf("Дочерний процесс создан и сейчас запустит новую
программу\n");
      sleep(5);
      execlp(argv[1], 0);
      perror("Не удалось запустить новую программу");
      break;
    default:
      sleep(10);
       printf("Родительский процесс завершается\n");
  }
  return 0;
}
```

Программа, приведенная в листинге 1.4, использует функцию `fork` для "клонирования" родительского процесса. После вызова `fork` в системе выполняется уже не один, а два процесса, различающихся только значением, возвращенным функцией `fork`. Далее дочерний процесс, которому функция `fork` вернула значение 0, запускает новую программу при помощи функции `execlp`, причем новая программа замещает запустивший ее процесс. Обратите внимание, что мы не проверяем значение, возвращенное функцией `execlp`. Если прежняя программа продолжает выполняться после вызова этой функции, значит, произошла ошибка. Родительский же процесс выполняется дальше и завершается стандартным способом.

Более подробные сведения о функциях `fork`, `execlp` и других функциях `glibc` можно найти в справочной системе man. Мы же коснемся другого вопроса, который скорее всего возникает у программистов, начинающих писать программы на C/C++ в системе Linux, а именно вопроса о том, как определить, какая именно библиотека экспортирует ту или иную функцию. К сожалению, общего ответа на этот вопрос дать нельзя. Справочная система предоставляет информацию только о заголовочных файлах, в которых объявлены функции, но не о библиотеках, которые эти функции содержит. Если библиотека экспортирует таблицу имен, список ее функций можно получить с помощью команды `mn`. Однако для большинства библиотек ОС Linux такая таблица недоступна. Часто имя библиотеки содержит общие элементы с именем заголовочного файла, в котором объявлены ее функции. Например, функции, объявленные в файле vorbisfile.h, экспортируются библиотекой libvorbisfile.so.

Рассмотрим еще один пример консольного приложения, на этот раз на языке Delphi Language.

Листинг 1.5. Консольное приложение для ОС Linux на языке Delphi Language

```
program cat;

{$APPTYPE CONSOLE}

uses
  Libc;

const
  BUF_SIZE = $1000;

var
  fd, len : Integer;
  buf : array[0..BUF_SIZE-1] of Byte;

begin
  if ParamCount <> 1 then
  begin
    printf('Команда: %s имя_файла'+Char(10), @ParamStr(0)[1]);
    __exit(0);
  end;
  fd := open(@ParamStr(1)[1], O_RDONLY);
  if fd < 0 then
  begin
    perror(@ParamStr(1)[1]);
    __exit(1);
  end;
  len := __read(fd, buf[0], BUF_SIZE);
  while len > 0 do
  begin
    __write(stdout, buf[0], len);
    len := __read(fd, buf, BUF_SIZE);
  end;
  __close(fd);
  putchar(10);
  __exit(0);
end.
```

Программа cat (листинг 1.5) является аналогом одноименной утилиты ОС Linux, т. е. эта программа распечатывает в стандартный поток вывода содержимое файла, переданного ей в качестве параметра. Для того чтобы продемонстрировать особенности и возможности программирования консольных Pascal-приложений в Linux, в этой программе вместо стандартных функций языка Pascal используются их `glibc`-аналоги. Получившаяся в результате программа занимает на диске всего лишь 16 Кбайт, и вряд ли в Delphi можно добиться большего.

В Kylix Delphi функции библиотеки `glibc` объявляются в модуле `Libc`. Введенная в Pascal от Borland директива `varargs` позволяет импортировать функции C с переменным числом параметров. С помощью этой директивы модуль `Libc` импортирует, например, функцию `printf`:

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

Не следует забывать, однако, что директива `varargs` предназначена только для импортирования функций из внешних модулей, написанных на C, и создать процедуру с переменным числом параметров на языке Pascal с ее помощью не удастся.

Обратите также внимание, что метод подсчета параметров переменной ParamCount не совпадает с методом, принятым в C/C++. ParamCount не учитывает строку запуска программы, хотя функция ParamStr позволяет получить доступ к этой строке.

По неизвестным причинам импортируемая функция printf игнорирует спецификатор \n, так что приходится "вручную" добавлять код 10 в конце строки.

Имена некоторых функций из модуля `Libc` отличаются от своих прототипов из библиотеки `glibc` наличием префикса из двух символов "нижний дефис". Необходимость переименования идентификаторов вызвана их конфликтами с именами некоторых функций, операторов и типов из стандартных модулей Delphi Language. Вот список идентификаторов `glibc`, переименованных подобным образом:

`chdir`, `mkdir`, `rmdir`, `rename`, `exit`, `truncate`, `random`, `EOF`, `close`, `time`, `abort`, `strcat`, `strlen`, `abs`, `div`, `raise`, `index`, `sleep`, `flock`, `timercmp`, `read`, `write`

Следует отметить, что, несмотря на все достоинства программы cat, у нее есть один существенный недостаток — она непереносима. Кроссплатформенная версия программы cat (размер исполнимого файла которой будет примерно на 40 Кбайт превышать размер исполнимого файла Linux-специфичной программы) должна выглядеть следующим образом (листинг 1.6).

Листинг 1.6. Кросс-платформенное консольное приложение на языке Delphi Language

```
program catp;
{$APPTYPE CONSOLE}
uses
  SysUtils;
const
  BUF_SIZE = $1000;

var
  F : file of Byte;
  buf : array[0..BUF_SIZE-1] of Byte;

begin
 if ParamCount <> 1 then
 begin
   WriteLn('Команда: ', ParamStr(0), ' имя_файла');
   Halt(0);
 end;
 System.Assign(F, ParamStr(1));
try
  System.Reset(F);
except
  WriteLn(ErrOutput, 'Невозможно открыть файл ', ParamStr(1));
  Halt(1);
end;
len := BlockRead(F, buf[0], BUF_SIZE);
 while len > 0 do
 begin
   BlockWrite(Output, buf[0], len);
   len := BlockRead(F, buf[0], BUF_SIZE);
 end;
 System.Close(F);
 WriteLn;
end.
```

Обратите внимание на переменную `ErrOutput`. Эта переменная типа `Text` позволяет выводить данные в стандартный поток сообщений об ошибках. Наличие специального потока для вывода сообщений об ошибках — весьма

полезная возможность ОС Linux. Обычно поток ошибок и поток вывода направляют данные на одно и то же устройство (терминал), однако в ряде случаев, например, при перенаправлении вывода одной программы на ввод другой, эти потоки следует разделять. Переменная `ErrOutput` определена и в Delphi для ОС Windows, так что ее можно использовать в кросс-платформенных программах.

В заключение этого раздела следует коснуться одного весьма важного вопроса, который мы до сих пор не рассматривали. Речь идет об обработке ошибок, возникающих при вызове функций Linux API. Большинство функций `glibc` (как, впрочем, и многих других функций Linux) в случае ошибки возвращают значение −1, `NULL` или `EOF` (если это значение не имеет смысла для данной функции). Для того, чтобы получить более подробную информацию о возникшей ошибке, следует использовать специальные функции, объявленные в заголовочном файле errno.h и модуле `Libc`. Прежде чем рассмотреть эти функции, полезно будет разобрать механизм передачи данных об ошибках в библиотеке `glibc`. В библиотеке `glibc` существует переменная `errno` типа `int`. В случае возникновения ошибки при вызове функции `glibc` этой переменной присваивается значение, идентифицирующее ошибку. Переменная `errno` всегда содержит код последней ошибки, возникшей при вызове функций `glibc`. Таким образом, после очередного вызова функции переменная `errno` может содержать код ошибки, даже если вызов данной функции прошел успешно. Простая проверка значения `errno` может вызвать путаницу, особенно в той ситуации, когда переменная является единственным источником информации об ошибке. Для решения этой проблемы следует присваивать переменной `errno` значение 0 перед вызовом интересующей функции.

Переменная `errno`, как и константы, идентифицирующие коды ошибок, объявлены в файле errno.h. В модуле `Libc` также объявлены функции обработки ошибок, но здесь все несколько сложнее. Модули Pascal не могут импортировать переменные из библиотек ОС Linux, поэтому в модуле `Libc` переменная `errno` — это функция:

```
function errno: error_t;
```

Результатом функции является текущее значение переменной `errno`. А как же присвоить значение переменной `errno`? Для этого в модуль `Libc` импортируется функция `__errno_location`:

```
function __errno_location: PInteger; cdecl;
```

Эта функция возвращает указатель на переменную `errno`.

В одном из примеров мы уже столкнулись с функцией `perror`. Эта функция, объявленная в файлах stdio.h и модуле `Libc`, позволяет выводить на консоль строку, содержащую короткое сообщение о последней зафиксированной ошибке.

Файловая система Linux, разграничение прав пользователей

"В ОС Linux нет диска С". Эта особенность файловой системы ОС Linux, одна из первых, о которой узнают начинающие пользователи, безусловно не является самым важным различием файловых систем ОС Linux и ОС Windows. Подробное описание файловой системы ОС Linux выходит за рамки этой книги, ограничимся здесь перечнем основных ее особенностей, понимание которых важно для программиста.

ОС Linux является многопользовательской операционной системой с полной схемой файловых разрешений. Это означает, что ОС Linux не только имеет возможность реализовать одновременный доступ пользователей к файлам, но и предоставляет разные права доступа к файлам для разных пользователей. Под правами доступа понимается набор разрешенных операций над файлом. К возможным операциям относятся чтение файла, изменение и удаление (запись) файла и исполнение файла (эта операция имеет смысл для исполнимых файлов). Система разрешений для файлов Linux основана на системе идентификации пользователей.

Каждый пользователь Linux-системы имеет по меньшей мере два идентификатора (ID): *идентификатор пользователя* (uid) и один или более *идентификаторов групп* (gid). Каждый идентификатор пользователя является уникальным в масштабах системы. Он назначается определенному пользователю и содержится в файле /etc/passwd. Каждый пользователь входит, по меньшей мере, в одну группу. Первичный идентификатор группы каждого пользователя также хранится в файле /etc/passwd, а последующие — в файле /etc/group.

Все файлы ОС Linux имеют владельца, в качестве которого может выступать как отдельный пользователь, так и группа пользователей. Как правило, именно владелец файла определяет права доступа к файлу.

Для каждого файла определено три группы прав доступа: для владельца файла, для пользователей, входящих в основную группу владельца, и для всех остальных пользователей. В каждой группе можно указать три разрешения — на чтение, запись и исполнение файла. Таким образом, набор доступных пользователю операций над файлом определяется, во-первых, тем, какая из трех групп разрешений применима к пользователю (является ли пользователь владельцем файла, входит ли он в группу владельца или нет), а, во-вторых, тем, какие операции над файлом разрешены в соответствующей группе прав доступа.

Права доступа к файлам описываются либо тремя семиричными (трехразрядными двоичными) числами, где каждый бит соответствует одному из возможных разрешений, либо девятью символами из набора rwx (read, write, execute) и символами прочерка для не установленных разрешений.

В табл. 1.2 приводится структура разрешений для файла (эту структуру часто называют *маской разрешений*).

Таблица 1.2. *Структура описания прав доступа для файла*

Пользователь			**Группа**			**Остальные**		
Чтение	Запись	Исполнение	Чтение	Запись	Исполнение	Чтение	Запись	Исполнение
r	w	x	r	w	x	r	w	x

Например, если у владельца файла есть права на чтение, запись и исполнение, у группы — на чтение и запись, а у всех остальных — только на чтение, маска разрешений для этого файла выглядит как 764 или rwxrw-r--.

Кроме указанных девяти битов маска разрешений включает еще три бита: setuid, setgid и sticky. Речь об этих битах пойдет в следующем разделе, посвященном процессам.

Права доступа к файлам ограничиваются правами доступа к каталогам, в которых расположены файлы (в Linux каталоги тоже являются файлами). Даже если маска разрешений файла равна -rwxrwxrwx (777), к этому файлу не смогут обратиться те пользователи, у которых нет прав доступа к каталогу, в котором размещен файл. Поэтому, установив для каталога разрешения rwx------ (700), можно легко заблокировать все файлы данного каталога и всех его подкаталогов.

Возникает законный вопрос: какой смысл устанавливать права доступа для владельца файла, если владелец всегда может изменить любые права? В данном случае речь идет не о защите, а об удобстве. Например, нет смысла присваивать право на исполнение файлу, не являющемуся исполнимым, т. е. не являющимся двоичным файлом программы или файлом сценария.

Другой особенностью файловой системы ОС Linux является широкое использование *виртуальных файловых систем*. Под виртуальными файловыми системами в данном контексте понимаются иерархии каталогов и файлов, не связанные с каким-либо физическим устройством. Важным примером файловой системы этого типа является виртуальная файловая система /proc.

Файловая система /proc позволяет получить самую различную информацию о процессах, запущенных в системе, списке активных модулей ядра, статистике использования оперативной памяти и некоторых других параметрах системы. Неудивительно, что приложения, в которых необходимо отслеживать состояние системы, очень часто обращаются к файловой системе /proc. В каталоге /proc содержится целый ряд виртуальных каталогов и файлов. Многие файлы обновляются динамически по мере изменения состояния системы. В табл. 1.3 приводятся наиболее важные элементы файловой системы /proc.

Таблица 1.3. *Наиболее важные элементы файловой системы /proc*

Элемент	Тип	Содержание
cpuinfo	Файл	Полная информация о центральном процессоре
filesystems	Файл	Список поддерживаемых файловых систем
ide	Каталог	Сведения об имеющихся IDE-дисках
meminfo	Файл	Сведения об оперативной памяти
modules	Файл	Сведения о модулях ядра, загруженных в настоящий момент
net	Каталог	Сведения о состоянии сетевой подсистемы
scsi	Каталог	Сведения об имеющихся SCSI-устройствах
sys	Каталог	Множество сведений о системе
sysvipc	Каталог	Сведения об объектах межпроцессного взаимодействия System V

Кроме этого файловая система /proc предоставляет сведения о процессах, запущенных в системе. Данные о каждом процессе хранятся в подкаталоге, имя которого соответствует численному значению *идентификатора процесса* (pid). Идентификатор процесса является числом, выделяемым системой каждому созданному процессу. В каждый данный момент времени в системе может быть только один процесс с данным идентификатором. В подкаталоге процесса находятся несколько файлов и подкаталогов, из которых можно почерпнуть данные о различных аспектах выполняемого процесса.

В табл. 1.4 приведены элементы подкаталога процесса.

Таблица 1.4. *Элементы подкаталога процесса файловой системы /proc*

Элемент	Тип	Содержание
cmdline	Файл	Командная строка, использовавшаяся при запуске процесса
cwd	Символическая ссылка	Указывает на каталог процесса
environ	Файл	Список переменных окружения для данного процесса
exe	Символическая ссылка	Указывает на файл, хранящий образ процесса
Fd	Каталог	Ссылки на файлы, используемые процессом
root	Гибкая Ссылка	Указывает на корень файловой системы процесса
stat	Файл	Различные сведения о процессе

Другой важный пример виртуальной файловой системы ОС Linux — файловая система /dev, в которой содержатся файлы, соответствующие различным устройствам. В ОС Linux каждое устройство рассматривается как файл. Управление устройствами осуществляется также, как и управление файлами, при помощи системного вызова ioctl, а чтение и запись данных в устройство — путем чтения и записи данных в соответствующий файл. В качестве примера можно указать файл /dev/cdrom, соответствующий приводу для чтения компакт-дисков и файл /dev/dsp, соответствующий устройству волнового вывода звуковой карты.

Еще одна важная особенность файловой системы Linux заключается в возможности создания *ссылок на файл* (file links). Ссылки могут быть *жесткими* (hard links) и *символическими* (symbolic links). Жесткие ссылки фактически представляют собой дополнительные имена файлов (таким образом, в ОС Linux у одного файла может быть несколько имен). Символические ссылки — это особые файлы, содержащие путь к файлу, на который они ссылаются. Имя символической ссылки часто отличается от имени файла, на который она ссылается. Символическая ссылка может выступать заместителем файла во многих важных операциях файловой системы, прежде всего, при открытии файла. Символические ссылки играют важную роль при использовании разделяемых библиотек ОС Linux, и мы подробно рассмотрим их в соответствующем разделе.

В стандартном Pascal, разумеется, отсутствуют средства управления разрешениями файлов, равно как и средства, подобные функции `ioctl`. Однако подобные средства есть в библиотеке `glibc`, которую импортирует модуль `Libc`. Рассмотрим несколько практических примеров использования дополнительных возможностей файловой системы ОС Linux.

Первый пример — программа на Delphi Language, позволяющая считывать и устанавливать разрешения для файлов. На рис. 1.2 представлена главная форма программы.

Для того чтобы выбрать файл для просмотра и редактирования прав доступа, следует нажать кнопку **Выбрать** (компонент `Button2` в тексте программы). После выбора файла элементы управления CheckBox принимают состояния, соответствующие маске разрешений выбранного файла. Для того, чтобы изменить разрешения файла, необходимо изменить состояния соответствующих элементов CheckBox и нажать кнопку **Установить** (компонент `Button3` в тексте программы). Наша программа не позволяет изменять маску разрешений для файла, если идентификатор пользователя не совпадает с идентификатором владельца файла (если только пользователь — не root).

В листинге 1.7 приводится текст модуля программы.

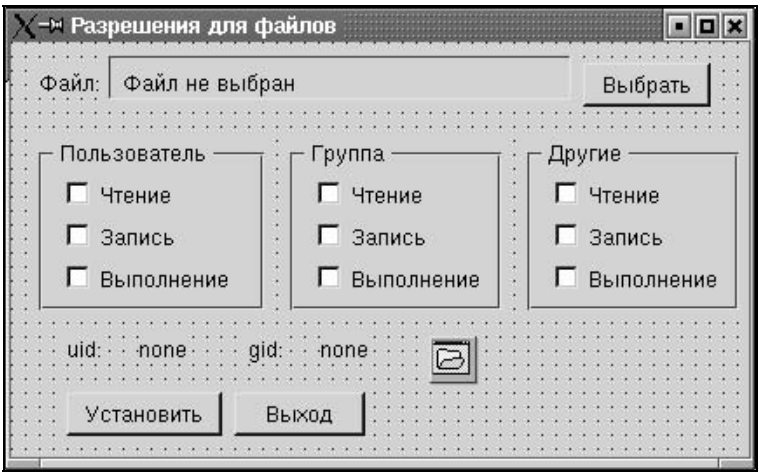


Рис. 1.2. Форма программы-примера

Листинг 1.7. Программа чтения и установки разрешений для файлов

```
unit Main;


interface


uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms,
QDialogs,
  QStdCtrls, Libc, QExtCtrls;


type
  TForm1 = class(TForm)
    OpenDialog1: TOpenDialog;
    GroupBox1: TGroupBox;
    GroupBox2: TGroupBox;
   { Элементы для отображения/установки прав пользователя }
   UserRead: TCheckBox;
    UserWrite: TCheckBox;
    UserExecute: TCheckBox;
    { Элементы для отображения/установки прав группы }
    GroupRead: TCheckBox;
    GroupWrite: TCheckBox;
```

```
    GroupExecute: TCheckBox;
  { Элементы для отображения/установки прав остальных пользователей}
   OthersRead: TCheckBox;
    OthersWrite: TCheckBox;
    OthersExecute: TCheckBox;
    GroupBox3: TGroupBox;
    Label1: TLabel;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    FileLabel: TLabel;
    Panel1: TPanel;
    Label3: TLabel;
    UidLabel: TLabel;
    Label5: TLabel;
    GidLabel: TLabel;
procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
  { Private declarations }
{ Процедура, читающая разрешения для файла }
      procedure GetPermissions;
{ Процедура, записывающая разрешения для файла }
      procedure SetPermissions;
  public
      { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
  if (OpenDialog1.Execute) then
  begin
    filelabel.Caption := OpenDialog1.FileName;
    GetPermissions;
  end;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
   close;
end;

procedure TForm1.GetPermissions;
var
  perms: Cardinal;
  buf: TStatBuf;
  filename: String;
begin
  UserRead.Checked := false;
  UserWrite.Checked := false;
  UserExecute.Checked := false;
  GroupRead.Checked := false;
  GroupWrite.Checked := false;
  GroupExecute.Checked := false;
  OthersRead.Checked := false;
  OthersWrite.Checked := false;
  OthersExecute.Checked := false;
  filename := FileLabel.Caption;
  if stat(PChar(filename), buf) = -1 then
  begin
    ShowMessage('Ошибка при чтении разрешений файла.');
    Exit;
  end;
  perms := buf.st_mode;
  UidLabel.Caption := IntToStr(buf.st_uid);
  GidLabel.Caption := IntToStr(buf.st_gid);
  if (perms and S_IRUSR <> 0) then
  UserRead.Checked := true;
```

```
  if (perms and S_IWUSR <> 0) then
  UserWrite.Checked := true;
  if (perms and S_IXUSR <> 0) then
  UserExecute.Checked := true;
  if (perms and S_IRGRP <> 0) then
  GroupRead.Checked := true;
  if (perms and S_IWGRP <> 0) then
  GroupWrite.Checked := true;
  if (perms and S_IXGRP <> 0) then
  GroupExecute.Checked := true;
  if (perms and S_IROTH <> 0) then
  OthersRead.Checked := true;
  if (perms and S_IWOTH <> 0) then
  OthersWrite.Checked := true;
  if (perms and S_IXOTH <> 0) then
  OthersExecute.Checked := true;
end;

procedure TForm1.SetPermissions;
var
  perms: Cardinal;
  filename: String;
  buf : TStatBuf;
  uid : __uid_t;
begin
  filename := FileLabel.Caption;
  if stat(PChar(filename), buf) < 0 then
  begin
    ShowMessage(' Ошибка при чтении разрешений файла.');
    Exit;
  end else
  begin
    uid := getuid;
    if (buf.st_uid <> uid) and (uid > 1) then
    begin
    { Идентификаторы пользователя и владельца файла не совпадают,
      и вы — не root }
```

```
      ShowMessage('У вас нет прав на изменение маски разрешений↵
      для этого файла.');
      Exit;
    end;
  end;
  perms := 0;

  if (UserRead.Checked) then
  perms := perms or S_IRUSR;
  if (UserWrite.Checked) then
  perms := perms or S_IWUSR;
  if (UserExecute.Checked) then
  perms := perms or S_IXUSR;
  if (GroupRead.Checked) then
  perms := perms or S_IRGRP;
  if (GroupWrite.Checked) then
  perms := perms or S_IWGRP;
  if (GroupExecute.Checked) then
  perms := perms or S_IXGRP;
  if (OthersRead.Checked) then
  perms := perms or S_IROTH;
  if (OthersWrite.Checked) then
  perms := perms or S_IWOTH;
  if (OthersExecute.Checked) then
  perms := perms or S_IXOTH;
  if chmod(PChar(filename), perms) < 0 then
  begin
    ShowMessage('Ошибка при установке разрешений для файла.');
    getPermissions;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  SetPermissions;
  GetPermissions;
end;
end.
```

Маску разрешений файла можно получить при помощи вызова функции `stat` из модуля `Libc` (маска разрешений содержится в поле `st_mode` структуры `TStatBuf`, возвращенной функцией `stat`). Кроме маски разрешений функция `stat` возвращает множество другой полезной информации о файле (в том числе, размер файла, дату создания и идентификатор владельца). Установка маски разрешений выполняется при помощи функции `chmod`. Следует отметить, что в ОС Linux имена многих системных функций совпадают с именами аналогичных команд оболочки. Для установки битов маски разрешений можно использовать константы `S_I*`. Перечень этих констант приводится в табл. 1.5.

Таблица 1.5. *Константы битов маски разрешений для файла*

Константа	Пояснение
`S_IRUSR`	Право чтения для пользователя
`S_IWUSR`	Право записи для пользователя
`S_IXUSR`	Право выполнения для пользователя
`S_IRGRP`	Право чтения для группы
`S_IWGRP`	Право записи для группы
`S_IXGRP`	Право выполнения для группы
`S_IROTH`	Право чтения для остальных пользователей
`S_IWOTH`	Право записи для остальных пользователей
`S_IXOTH`	Право выполнения для остальных пользователей

С помощью функции `getuid` программа получает идентификатор пользователя и сравнивает его с идентификатором владельца файла. Если пользователь пытается изменить маску разрешений файла, владельцем которого он не является, программа выдает сообщение об ошибке (*подробную информацию о функции* `getuid` *см. в разд. "Процессы и сигналы" этой главы*).

Теперь рассмотрим пример использования виртуальной файловой системы /proc. Следующая программа (листинги 1.8 и 1.9), написанная на C++, выводит список своих переменных окружения.

Листинг 1.8. Файл MainFrm.h, объявление главного класса приложения

```
//---------------------------------------------------------------------------

#ifndef MainFrmH
#define MainFrmH
//---------------------------------------------------------------------------
```

```
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <QActnList.hpp>
#include <QComCtrls.hpp>
#include <QDialogs.hpp>
#include <QExtCtrls.hpp>
#include <QImgList.hpp>
#include <QMenus.hpp>
#include <QStdActns.hpp>
#include <QTypes.hpp>
//---------------------------------------------------------------------------

class TMainForm : public TForm
{
__published:  // IDE-managed Components
        TMemo *EnvMemo; // Компонент для вывода строк окружения
        TPanel *Panel1;
        TLabel *Label1;
        TButton *ShowButton; // Кнопка "Показать"
        void __fastcall ShowButtonClick(TObject *Sender);
private:      // User declarations
public:// User declarations
        __fastcall TMainForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------------------------------------
#endif
```

Листинг 1.9. Файл MainFrm.cpp — определение методов класса `TMainForm`

```
//---------------------------------------------------------------------------

#include <clx.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
```

```
#include <fcntl.h>
#pragma hdrstop

#include "MainFrm.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TMainForm *MainForm;
AnsiString sUntitled = "Untitled";

//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
        : TForm(Owner)
{
}

//---------------------------------------------------------------

void __fastcall TMainForm::ShowButtonClick(TObject *Sender)
{
  #define BUF_SIZE 0x1000
  char buf[BUF_SIZE];
  int fd, len, i;
  char fn[64];
  AnsiString S;
  memset(fn, 0, 64);
  /* Определяем путь к файлу environ для текущего процесса */
  sprintf(fn, "/proc/%d/environ", getpid());
  fd = open(fn, O_RDONLY);
  if (fd < 0) {
    ShowMessage("Не удалось открыть файл переменных окружения");
    return;
  }
  len = read(fd, buf, BUF_SIZE-1);
  close(fd);
  buf[len] = 0;
  /* Заполняем компонент TMemo строками окружения */
  i = 0;
```

```
  while (i<len) {
    S = &buf[i];
    EnvMemo->Lines->Add(S);
    while (buf[i++] != 0);
  }
}
//---------------------------------------------------------------------
```

Обращаем внимание на то, как мы определяем путь к файлу environ. Как указывалось выше, файлы, содержащие информацию для данного процесса, хранятся в каталогах системы /proc, имена которых совпадают со значениями идентификаторов процессов. В методе `ShowButtonClick` мы сначала узнаем идентификатор нашего процесса (функция `getpid`), а затем с помощью функции `sprintf` формируем строку, содержащую полный путь к файлу environ. Впрочем, можно поступить проще: ссылка /proc/~self указывает на каталог текущего процесса. Не стоит забывать, что /proc — виртуальная файловая система, и содержимое ее файлов зависит не только от состояния системы в данный момент, но и от того, какой процесс к ним обращается. Строки окружения разделяются в файле environ символами конца строки (0x0).

На рис. 1.3 приводится результат работы программы.

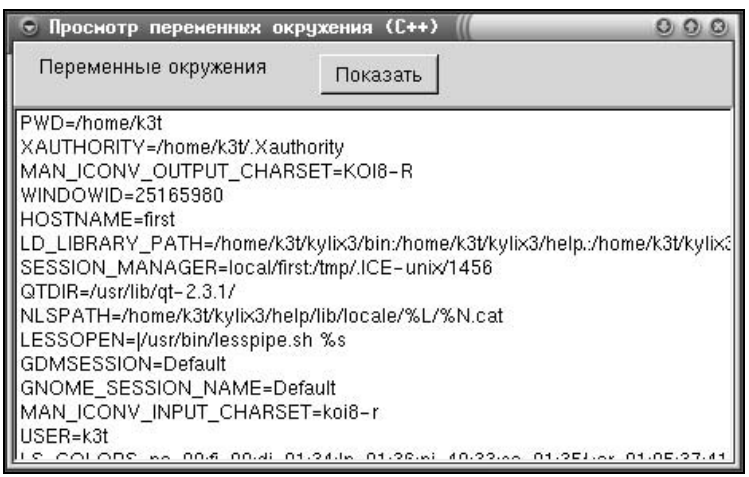


Рис. 1.3. Программа, читающая строки окружения

В заключение этого раздела рассмотрим еще один пример работы с файловой системой ОС Linux. В процессе программирования часто возникает задача перечислить имена всех файлов, хранящихся в данном каталоге. Эту задачу можно решить стандартными средствами Delphi Language `FindFirst/FindNext`. Достоинством такого подхода является

переносимость, а недостатком — низкое быстродействие, особенно заметное тогда, когда нам приходится сканировать каталоги с очень большим числом файлов. Приводимая в листинге 1.10 процедура `ListFiles` работает гораздо быстрее за счет использования специальной функции `scandir` из модуля `Libc`.

Листинг 1.10. Процедура ListFiles

```
procedure ListFiles(const DirectoryPath : String; Files : TStringList);
var
  des : PPDirEnt;
  i, n : Integer;
function sel(de : PDirEnt) : Integer; cdecl;
begin
Result := 1;
end;
begin
  n := scandir(@DirectoryPath[1], des, sel, alphasort);
  if n < 0 then
  raise Exception.Create(Format('Ошибка при чтении каталога %s',
@DirectoryPath[1]));
  for i:=0 to n-1 do
  begin
    Files.Add(String(des^.d_mane));
    Inc(des);
  end;
end.
```

Процедуре `ListFiles` передается два параметра: путь к сканируемому каталогу и ссылка на заранее созданный экземпляр объекта `TStringList`, в который будут записаны имена элементов каталога. Функция `scandir` создает список элементов указанного каталога. Функции `scandir` необходимо передать указатель на функцию обратного вызова, которая, получая данные об очередном элементе, принимает решение, включать ли этот элемент в результирующий список. В нашем примере это функция `sel`. Если при очередном вызове функция `sel` вернет значение 0, соответствующий элемент каталога в конечный список включен не будет. Объявляя функцию `cb`, мы впервые столкнулись с необходимостью использовать формат вызова `cdecl`. Этот формат используется по умолчанию во всех библиотеках Linux, также как в Windows используется `stdcall`. Последний параметр `scandir` — функция сортировки элементов каталога. Мы используем `glibc`-функцию `alphasort`, сортирующую элементы в лексикографическом порядке.

Данные об элементах каталога передаются в структурах `dirent`. Кроме имени файла структура `dirent` содержит информацию о типе файла (обычный файл, каталог, файл специального типа, ссылка и т. п.), а также некоторые другие данные. Возможность получить сведения о типе файла является еще одним преимуществом использования функции `scandir`. Функция `scandir` возвращает число элементов в созданном списке.

Другой вариант решения той же задачи в glibc связан с использованием функций `opendir`, `readdir` и `closedir`. Напомним, что в ОС Linux каталоги являются файлами и функции `opendir`, `readdir` и `closedir` очень похожи на функции доступа к обычным файлам. В листинге 1.11 приводится вариант функции `ListFiles` для C++ с использованием указанных выше функций.

Листинг 1.11. Функция `ListFiles`

```
void ListFiles(String DirectoryPath, TStringList * Files) {
  struct dirent *ep;
  DIR *dp = opendir(&DirectoryPath[1]);
  if (dp != NULL)
  {
     while (ep = readdir (dp))
     {
        AnsiString S = ep->d_name;
        Files->Add(S);
     }
     closedir (dp);
  }
  else throw "Ошибка при открытии каталога";
}
```

Функции и типы, используемые в функции `ListFiles`, объявлены в файле dirent.h.

Процессы и сигналы

Процессы

В предыдущих разделах этой главы мы уже познакомились с некоторыми характеристиками процессов Linux. С помощью идентификатора процесса (pid) мы получали различные данные о процессе. Если мы немного изменим код программы чтения переменных окружения (листинг 1.9) и попробуем с ее по-

мощью получить данные о других процессах, то обнаружим, что данные доступны не для всех процессов, перечисленных в каталоге `/proc`. Дело в том, что для процессов Linux действует система привилегий, тесно связанная с системой прав доступа к файлам. Следует помнить, что в Linux многие важные объекты операционной системы представляются в виде файлов, поэтому система разграничения прав доступа к файлам фактически определяет разграничение прав доступа ко всем основным ресурсам системы. Процесс Linux может использовать только те ресурсы ОС, на доступ к которым у него есть права. Это правило относится к файлам на диске, объектам межпроцессного взаимодействия, файлам устройств, системным вызовам и т. п. Кроме того, между процессами, запущенными с привилегиями root, и процессами, запущенными с привилегиями обычных пользователей, существуют важные различия, касающиеся прав доступа к файловой системе и установки приоритетов процесса.

Как же реализована система привилегий для процессов Linux?

Как и в случае системы разграничения прав доступа к файлам, основой системы привилегий процессов являются идентификаторы пользователя и группы. Это естественно, ведь пользователь осуществляет доступ к файлам при помощи какого-либо процесса, и права пользователя на доступ к файлам с точки зрения системы являются правами процесса, запущенного пользователем. Обычно процесс получает реквизиты uid и gid, соответствующие идентификаторам пользователя, запустившего этот процесс. Если пользователь, запустивший программу, является владельцем исполняемого файла, реквизиты процесса и реквизиты владельца файла не совпадают. Исключение составляют программы, для которых установлены биты setuid и setgid. Процессы, запускаемые из таких файлов, получают не реквизиты запускающего их пользователя, а реквизиты владельца файла.

Текущие реквизиты процесса называются *эффективными (effective)* и составляют *паспорт процесса* (в англоязычной литературе используется термин *persona of a process)*. Кроме эффективных, у процесса могут быть *реальные (real)* реквизиты, соответствующие uid и gid пользователя, создавшего процесс. Если пользователь, владеющий процессом, принадлежит нескольким группам пользователей, у процесса появляется ряд *дополнительных групп*.

В какой ситуации эффективные и реальные реквизиты процесса могут не совпадать? Это происходит, например, в том случае, когда в ходе выполнения процесс меняет свой паспорт. Примером процесса, меняющего паспорт, является программа login, осуществляющая регистрацию пользователей в системе. При подключении терминала программа login запускается с uid root. Это естественно, ведь программа, выполняющая регистрацию пользователей, должна иметь доступ к файлам, принадлежащим суперпользователю. После того, как пользователь осуществляет идентификацию и авторизацию в системе, программа login меняет паспорт root на паспорт пользователя (если, конечно, пользователь не зарегистрировался как root). Смена паспорта может потребоваться не только программе login. Другим примером приложений такого типа являются программы, обычно работаю-

щие с правами запустившего их пользователя, но иногда требующие прав суперпользователя. К этому классу программ относятся программы настройки параметров системы.

С получением идентификатора владельца процесса мы уже сталкивались в программе установки разрешений для файлов (листинг 1.7). Здесь же мы рассмотрим, каким образом процесс может изменить свой паспорт. Процессы, владельцем которых является root, могут изменить свой паспорт на любой другой, возможный в системе. Для этой цели служат glibc-функции setuid и setgid. Стоит отметить, что эти функции изменяют не только эффективные, но и реальные реквизиты процесса. Обычный пользователь может запустить процесс с реквизитами другого пользователя только в том случае, если у соответствующего двоичного файла установлены биты setuid и setgid. Наличие битов setuid и setgid — это еще одна характерная черта файловых систем Unix. Если у двоичного файла установлен бит setuid или setgid, то при запуске процесса из этого файла новый процесс получает эффективные реквизиты, соответствующие реквизитам пользователя-владельца файла. Реальные же реквизиты соответствуют реквизитам пользователя, запустившего процесс. Таким образом, если у файла программы, владельцем которой является root, установлены биты setuid и setgid, то при запуске этой программы обычным пользователем процесс получит эффективные реквизиты пользователя root, а значит и его права. Естественно, для того чтобы любой пользователь мог запускать программы, владельцем которых является root, суперпользователь должен разрешить исполнение файла другим пользователям.

Процесс, эффективные реквизиты которого отличаются от реальных, всегда имеет возможность преобразовать их, т. е. процесс, запущенный обычным пользователем с эффективными реквизитами суперпользователя, всегда может установить такие же реквизиты, как у запустившего его пользователя. Если мы пишем программу, предназначенную для запуска обычным пользователем с эффективными реквизитами суперпользователя, следует предусмотреть переход к реквизитам обычного пользователя сразу после того, как полномочия суперпользователя перестанут быть необходимыми.

Еще одной важной характеристикой процесса является *приоритет*, который можно определить, как долю машинного времени, которое система выделяет процессу. В ОС Linux приоритеты процессов задаются числами от −20 до 19, причем чем меньше число, тем выше приоритет процесса. Отрицательные значения приоритетов доступны только суперпользователю. Процессы обычных пользователей при запуске получают приоритет со значением 0, который затем может только понижаться, т. е. значение приоритета будет увеличиваться.

Рассмотрим пример установки приоритета для приложения, написанного на Delphi Language. Диалоговое окно **Приоритет процесса** (рис. 1.4) будет по-

лезно в приложениях, выполняющих длительные по времени задачи, которые можно перевести в фоновый режим. Листинг 1.12 представляет собой текст модуля диалогового окна.

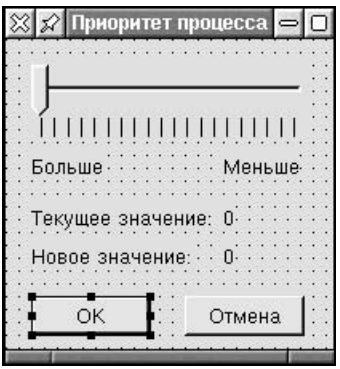


Рис. 1.4. Окно **Приоритет процесса**

Листинг 1.12. Модуль окна Приоритет процесса

```
unit SetPriority;

interface

uses
  Libc, SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms,
  QDialogs, QComCtrls, QStdCtrls, QExtCtrls, QTypes;

type
  TSetPriDLG = class(TForm)
    TrackBar1: TTrackBar;
    Label1: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    LabelCurVal: TLabel;
    LabelNewVal: TLabel;
    ButtonCancel: TButton;
    ButtonOK: TButton;
    procedure ButtonOKClick(Sender: TObject);
```

```
    procedure FormShow(Sender: TObject);
    procedure ButtonCancelClick(Sender: TObject);
    procedure TrackBar1Change(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  SetPriDLG: TSetPriDLG;

implementation

{$R *.xfm}

procedure TSetPriDLG.FormShow(Sender: TObject);
begin
  TrackBar1.Position := getpriority(PRIO_PROCESS, getpid());
  LabelCurVal.Caption := IntToStr(TrackBar1.Position);
  LabelNewVal.Caption := IntToStr(TrackBar1.Position);
end;

procedure TSetPriDLG.TrackBar1Change(Sender: TObject);
begin
  LabelNewVal.Caption := IntToStr(TrackBar1.Position);
end;

procedure TSetPriDLG.ButtonCancelClick(Sender: TObject);
begin
  Close;
end;

procedure TSetPriDLG.ButtonOKClick(Sender: TObject);
begin
  setpriority(PRIO_PROCESS, getpid(), TrackBar1.Position);
  Close;
end;
end.
```

Чтение и установка текущего приоритета процесса выполняются при помощи функций `getpriority` и `setpriority` (программисты, пишущие на языках C/C++, найдут декларации этих функций в файле sys/resource.h). Первый аргумент этих функций указывает тип объекта, для которого изменяется приоритет. Кроме приоритета процесса, можно изменять приоритеты пользователей и групп, если, конечно, у вас есть на это право. Второй аргумент функций `getpriority` и `setpriority` — идентификатор объекта. В нашем случае это идентификатор текущего процесса. Функция `getpriority` возвращает значение текущего приоритета для указанного объекта (или сообщение об ошибке, если это значение не может быть получено), а функция `setpriority` позволяет передать новое значение приоритета в качестве третьего аргумента. Важно отметить, что приоритет процесса, пониженный с помощью `setpriority`, не может быть снова повышен до прежнего значения.

Сигналы

Одним из простейших и одновременно важнейших средств обмена данными между процессами являются *сигналы*. Сигналы Linux (их не следует путать с сигналами Qt library, о которых речь пойдет в следующей главе) представляют собой программные прерывания, служащие для однонаправленной передачи сообщений от одного процесса другому.

В настоящее время в системе Linux определен 31 сигнал. При этом 29 сигналов зарезервированы, а 2 - предоставлены для использования программистами по своему усмотрению. В табл. 1.6 перечислены наиболее часто встречаемые сигналы Linux и дано их краткое описание.

Таблица 1.6. *Сигналы ОС Linux*

Обозначение	Описание
SIGABRT	Завершение процесса в результате вызова функции abort
SIGALRM	Сигнал таймера
SIGCHLD	Дочерний процесс завершился
SIGCONT	Возобновление выполнения приостановленного процесса. Не может игнорироваться
SIGBUS	Попытка освободить незанятую область памяти
SIGFPE	Ошибка при выполнении операции с плавающей точкой
SIGHUP	Отключение пользовательского терминала. Процессы-демоны обычно перезапускаются в ответ на этот сигнал

Таблица 1.6 *(окончание)*

Обозначение	Описание
SIGINT	Завершение процесса при помощи комбинации клавиш <Ctrl>+<C>
SIGIO	Сигнал готовности файлового дескриптора (асинхронный режим)
SIGKILL	Принудительное завершение процесса. Сигнал нельзя ни обрабатывать, ни игнорировать.
SIGPIPE	Канал связи между процессами разорван
SIGSEGV	Ошибка сегментации, нарушение прав доступа к оперативной памяти
SIGSTOP	Приостановка выполнения процесса. Сигнал не может игнорироваться.
SIGTERM	"Вежливое" завершение процесса. Сигнал может игнорироваться.
SIGURG	"Срочный" сигнал о поступлении данных (асинхронный режим)

Сигналы `SIGUSR1` и `SIGUSR2` предназначены для использования программистами.

Большинство сигналов посылается процессу системой при возникновении какой-либо чрезвычайной ситуации. Если процесс не установил обработчика таких сигналов, вызывается обработчик по умолчанию, который, как правило, завершает процесс.

Собственный обработчик сигналов может понадобиться программе по разным причинам. Например, в обработчиках сигналов завершения процесса можно разместить код, высвобождающий занятые программой ресурсы и удаляющий временные файлы. Если мы хотим использовать системные таймеры ОС Linux, нам понадобится обработчик сигнала `SIGALRM`. Мы также можем захотеть, чтобы наша программа игнорировала некоторые сигналы, например, сигнал, поступающий при нажатии комбинации клавиш <Ctrl>+<C>.

Назначение обработчика сигнала осуществляется при помощи `glibc`-функции signal. Эта функция объявляется в файле signal.h следующим образом:

```
sighandler_t signal (int signum, sighandler_t action)
```

Первый параметр этой функции — номер сигнала. Допустимым номерам сигналов соответствуют константы, аналогичные обозначениям сигналов из таблицы 1.6. Второй параметр функции `signal` — это функция обработчик сигнала. Данная функция будет вызываться всякий раз при поступлении сигнала и заменит обработчик, установленный по умолчанию. Заголовок функции обработчика сигнала должен выглядеть следующим образом:

```
void function_name (int signum)
```

Единственный параметр этой функции — номер сигнала (один и тот же обработчик может обрабатывать несколько сигналов). Программистам, пишущим на языке Delphi Language, следует помнить о том, что функция обработчик, как и все функции обратного вызова Linux, должна использовать формат вызова `cdecl`. В качестве второго аргумента функции `signal` можно указывать не только функцию обработчик, но и несколько специальных констант из файла system.h. Константа `SIG_DFL` сообщает функции `signal`, что для данного сигнала следует восстановить обработчик, используемый по умолчанию. Константа `SIG_IGN` указывает, что процесс должен игнорировать соответствующий сигнал (если это возможно). Если при вызове функции `signal` не произошло ошибки, она возвращает указатель на ранее установленный обработчик сигнала. Это значение можно использовать либо для восстановления прежнего обработчика, либо для вызова нескольких обработчиков по цепочке. В листинге 1.13 приводится фрагмент программы, обрабатывающей сигналы `SIGTERM` и `SIGINT`.

Листинг 1.13. Фрагмент кода обработки сигналов

```
#include <signal.h>
#include <stdio.h>
...
void OnExit(int signum)
{
  switch(signum)
  {
    case SIGTERM:
      printf("До свидания!\n");
      break;
    case SIGINT
      printf("Вы нажали Ctrl-C!\n");
      break;
  }
  signal(SIGTERM, SIG_DFL);
  signal(SIGINT, SIG_DFL);
  raise(signum);
}
int main()
{
  signal(SIGTERM, OnExit);
  signal(SIGINT, OnExit);
  ...
}
```

Стоит обратить внимание на восстановление обработчиков по умолчанию в конце функции `OnExit`, а также на вызов функции `raise` (напомним, что в модуле `Libc` для Delphi Language эта функция названа `__raise`), заставляющей процесс повторно послать переданный сигнал самому себе. Все это необходимо для корректной обработки сигналов завершения процесса. Без этих вызовов программа не завершилась бы при получении сигналов `SIGTERM` и `SIGINT`.

Для посылки сигнала другому процессу можно использовать функцию `kill`:

```
int kill (pid_t pid, int signum);
```

Первый параметр функции — идентификатор процесса, которому посылается сигнал, второй — номер сигнала. Функция возвращает значение 0, если передача сигнала прошла успешно, и 1 - в случае ошибки. Следует помнить, что процесс может посылать сигналы только тем процессам, уровень привилегий которых не превышает его собственный.

Более подробные сведения об обработке сигналов можно найти в документации по `glibc` и справочной системе man. В заключение этой темы отметим, что в программах, обладающих графическим интерфейсом, собственные обработчики сигналов следует применять с определенной осторожностью. Графические приложения сами устанавливают обработчики для многих сигналов Linux, и замена этих обработчиков может привести к нарушению корректной работы приложения. Кроме того, библиотеки программирования графических интерфейсов обычно предоставляют альтернативные средства обработки важных событий системы, более подходящие для использования в графическом приложении.

Потоки Linux и класс *TThread*

Как и ОС Windows, ОС Linux — многопоточная операционная система. Каждый процесс может создавать несколько "подпроцессов" — потоков, обладающих доступом ко всем его ресурсам. Можно предположить, что в ОС Linux потоки играют более существенную роль, чем в ОС Windows. Причина этого заключается в том, что, несмотря на все графические надстройки, ОС Linux во многом остается консольной ОС. Специфика консольных приложений позволяет широко использовать синхронное взаимодействие с другими частями системы. В синхронном режиме функция, осуществляющая взаимодействие, блокирует (приостанавливает) выполнение вызвавшего ее процесса до окончания транзакции. Для графических приложений, которые должны оперативно реагировать на многочисленные события, происходящие в графической системе, такой подход неприемлем.

Эту проблему можно решить несколькими способами:

- использовать асинхронный режим везде, где это возможно;

- разбить длительную операцию на небольшие этапы и обрабатывать внешние события в перерывах между ними;
- использовать потоки.

Последний подход представляется наиболее простым и профессиональным. Размещая код в потоке, можно смело применять синхронный режим, не беспокоясь о блокировании. В то же время основной процесс может по-прежнему реагировать на все события. Равноправный доступ потоков к ресурсам процесса требует организации системы разделения доступа к ресурсам.

В ОС Linux используется модель потоков POSIX. Функции управления потоками Linux экспортируются библиотекой libpthread.so и в большинстве своем имеют префикс `pthread_`. Программисты, пишущие на языках C/C++, должны использовать заголовочные файлы pthreadtypes.h и pthread.h, а те, кто предпочитает Delphi Language, найдут объявления этих функций в модуле Libc. Среди основных функций для работы с потоками следует указать функцию `pthread_create` (создание и запуск нового потока), `pthread_exit` (выход из завершенного потока) и `pthread_kill` (приостановка и последующий запуск потока). Функция `pthread_kill` очень похожа на функцию `kill`. Для управления потоками эта функция использует идентификатор потока, полученный от функции `pthread_create`, и сигналы `SIGSTOP` и `SIGCONT`.

Программисты, работающие в средах разработки от Borland, избавлены от необходимости заниматься низкоуровневым программированием потоков благодаря классу `TThread`. Этот класс инкапсулирует работу с потоками соответствующей операционной системы и не только упрощает использование потоков в приложении, но и делает многопоточные приложения платформо-независимыми. Чтобы добавить в Kylix-приложение поддержку потоков, необходимо создать класс-потомок класса `TThread`, переопределив в нем метод `Execute`, в который следует поместить код потока. Класс `TThread` содержит полный набор методов, необходимых для управления потоком. Хотя класс `TThread` и обеспечивает кросс-платформенную переносимость приложений, реализации этого класса в средах разработки для ОС Windows и ОС Linux не полностью идентичны. Различия касаются механизма задания приоритета потока. В Kylix свойство `Priority` класса `TThread` имеет тип `Integer` (а не `TThreadPriority`, как в средах для Windows). Кроме того, в Linux-версии класса `TThread` имеется свойство `Policy`, определяющее смысл свойства `Priority`. Значением свойства `Policy` может быть одна из констант `SCHED_RR`, `SCHED_FIFO` и `SCHED_OTHER`. В непривилегированных процессах можно использовать только последнее значение, при котором значение свойства `Priority` должно всегда быть равным нулю. Иначе говоря, при использовании потока `TThread` в обычном приложении приоритет потока однозначно связывается с приоритетом процесса, и единственный способ изменить его заключается в том, чтобы изменить приоритет процесса.

Взаимодействие между процессами

Термином *межпроцессное взаимодействие* (inter-process communication) обозначается передача данных между процессами в рамках одной системы. Ранее мы уже рассмотрели простейший механизм межпроцессного взаимодействия, основанный на передаче сигналов. Этот механизм редко используется для обмена данными между прикладными программами из-за малой информативности сигналов. Далее мы рассмотрим средства межпроцессного взаимодействия, позволяющие передавать между процессами любые объемы данных.

По сравнению с ОС Windows система Linux обладает более богатым набором механизмов передачи данных между процессами. Отчасти это объясняется тем, что средства, изначально разрабатывавшиеся в разных ветвях Unix, внедрялись затем в другие версии в целях обеспечения совместимости.

В этом разделе будут рассмотрены такие средства взаимодействия между процессами, как однонаправленные каналы, сокеты, сообщения, разделяемые блоки памяти и семафоры. Механизмы обмена данными, основанные на функциях графических оболочек, такие как буферы обмена и Drag and Drop, мы рассмотрим в следующей главе.

Однонаправленные каналы

Однонаправленные каналы (pipes, в русскоязычной литературе иногда применяется термин "конвейер"), как следует из их названия, позволяют передавать данные только в одном направлении. На уровне интерфейса программирования канал представляется двумя дескрипторами файлов, один из которых служит для чтения данных, а другой — для записи. Каналы не поддерживают произвольный доступ, т. е. данные из канала могут считываться только в том же порядке, в котором они записывались. Однонаправленные каналы могут быть *анонимными* (anonymous pipes) и *именованными* (named pipes, FIFOs). Главное различие между анонимными и именованными каналами заключается в том, что анонимные каналы предназначены для обмена данными между родительским и дочерним процессами, в то время как именованные каналы позволяют осуществлять обмен данными между процессами, не зависящими друг от друга.

Обычно работа с анонимными каналами состоит из трех этапов. Сначала с помощью функции `pipe` (которая объявлена в файле unistd.h и модуле `Libc`) создается анонимный канал. В качестве параметра функции `pipe` передается массив типа `int`, состоящий из двух элементов. В первом элементе массива функция возвращает дескриптор файла, служащий для чтения данных, а во втором — дескриптор для записи.

Затем с помощью функции `fork` создается дочерний процесс. Дочерний процесс наследует от родительского оба дескриптора, но так же, как и родительский процесс должен использовать только один из них. Направление передачи данных между родительским и дочерним процессами определяется тем, какой дескриптор будет использоваться родительским процессом, а какой — дочерним.

Рассмотрим программу `pipes.cpp`, иллюстрирующую использование анонимных каналов (листинг 1.14).

Листинг 1.14. Программа pipes.cpp

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
int main ()
{
  int pipedes[2];
  pid_t pid;
  pipe(pipedes);      // создаем канал
  /* дескриптор pipedes[0] открыт для чтения, а pipedes[1] — для записи
*/
  pid = fork();       // создаем два процесса
  if (pid > 0)
  {
    // родительский процесс
    char  *str = "String passed via pipe\n";
    write(pipedes[1], (void *) str, strlen(str) + 1);
    close(pipedes[1]);
  }
  else
  {
    // дочерний процесс
    char buf[1024];
   int len
   while ((len = read(pipdes[0], buf, 1024)) != 0)
   write(stdout, buf, len);
  close(pipdes[0])
  }
```

```
return 0;
}
```

Именованные каналы отличаются от анонимных наличием идентификатора. Для идентификации именованного канала на диске создается файл специального типа `pipe`. Следует помнить о том, что файлы именованных каналов являются средством передачи, а не хранения, данных. В частности, размер файла канала всегда остается равным нулю. Для файлов именованных каналов действуют те же правила контроля доступа, что и для обычных файлов Linux. Именованные каналы создаются функцией `mkfifo`. Первый параметр этой функции — строка, в которой передается имя файла, идентифицирующего канал, второй параметр — маска прав доступа к файлу. Функция `mkfifo` вызывается для создания файла только один раз. После создания файла канала оба приложения, участвующие в обмене данными, должны открыть этот файл. Одно приложение открывает файл канала для записи, другое для чтения.

В листинге 1.15 приводится исходный текст программы-сервера, создающей именованный канал и записывающей в него строку, а в листинге 1.16 — текст программы-клиента, читающей строку из канала.

Листинг 1.15. Программа-сервер

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
int main ()
{
  FILE * f;
  char * str = "String passed via FIFO";
  char * fname = "~/fifofile";
  /* создаем файл канала с маской доступа rw-rw-rw- */
  mkfifo(fname,  0xDB);
  f = fopen(fname, "w");   // открываем файл для записи
  fwrite((void*) str, strlen(str) + 1, 1, f);
  fclose(f);
  return 0;
}
```

Листинг 1.16. Программа-клиент

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
   FILE * f;
   f = fopen("~/fifofile", "r");  // открываем файл для чтения
   while (putchar(getc(f)) != '\0');
   putchar('\n');
   fclose(f);
   return 0;
}
```

Программу-сервер следует запускать первой. Отметим, что система синхронизирует процессы, связанные с каналами. Например, при вызове функции чтения или записи в канал выполнение процесса приостанавливается до тех пор, пока "на другом конце" канала не будет выполнена комплиментарная операция.

Функции *popen* и *pclose*

Среди средств работы с каналами особого внимания заслуживает функция `popen`. Эта функция запускает внешнюю программу и возвращает вызвавшему ее приложению указатель, связанный либо со стандартным потоком ввода, либо со стандартным потоком вывода запущенного процесса. Первый параметр функции `popen` — строка, содержащая команду, которая запускает внешнюю программу. Второй параметр определяет, какой из стандартных потоков (ввода или вывода) будет возвращен. Аргумент `"w"` соответствует потоку ввода запускаемой программы. В этом случае приложение, вызвавшее `popen`, записывает данные в поток. Аргумент `"r"` соответствует потоку вывода. Функцию `popen` можно безопасно использовать и в графическом приложении. Поток, открытый с помощью функции `popen`, должен быть закрыт функцией `pclose`. Напишем в Kylix C++ приложение-пример `PopenDemo` (листинги 1.17 и 1.18). Наше приложение позволяет пользователю вводить какую-либо команду ОС Linux и выводить результат выполнения команды в окно компонента `Memo`. На рис. 1.5 представлена форма приложения `PopenDemo`.

Листинг 1.17. Заголовочный файл главного модуля приложения PopenDemo

```
//---------------------------------------------------------------------------

#ifndef MainH
#define MainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <QExtCtrls.hpp>
//---------------------------------------------------------------------------
class TMainForm : public TForm
{
__published:  // IDE-managed Components
        TMemo *Memo; // Компонент для вывода результата команды
        TPanel *Panel1;
        TEdit *EditCommand; // Строка ввода команды
        TLabel *Label1;
        TButton *ButtonExec; // Кнопка "Выполнить"
        void __fastcall ButtonExecClick(TObject *Sender);
        void __fastcall EditCommandKeyDown(TObject *Sender, WORD &Key,
          TShiftState Shift);
private:      // User declarations
public:       // User declarations
        __fastcall TMainForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------------------------------------
#endif
//---------------------------------------------------------------------------

#include <clx.h>
#include <stdio.h>
#pragma hdrstop
```

Листинг 1.18. Исходный текст главного модуля приложения PopenDemo

```
#include "Main.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"

#define BUF_SIZE 0x1000

TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TMainForm::ButtonExecClick(TObject *Sender)
{
  char buf[BUF_SIZE];
  int len;
  AnsiString S = (AnsiString) EditCommand->Text;
  FILE * F = popen(&S[1], "r");
  Memo->Clear();
  len = fread(buf, 1, BUF_SIZE-1, F);
  if (len == 0) Memo->Text = "Ошибка!";
  while (len != 0)
  {
    buf[len] = 0;
    S = buf;
    Memo->Text = Memo->Text + S;
    len = fread(buf, 1, BUF_SIZE-1, F);
  }
  pclose(F);
}
```

```
//---------------------------------------------------------------------------
void __fastcall TMainForm::EditCommandKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
  if (Key == 4100) ButtonExecClick(this);
}
//---------------------------------------------------------------------------
```

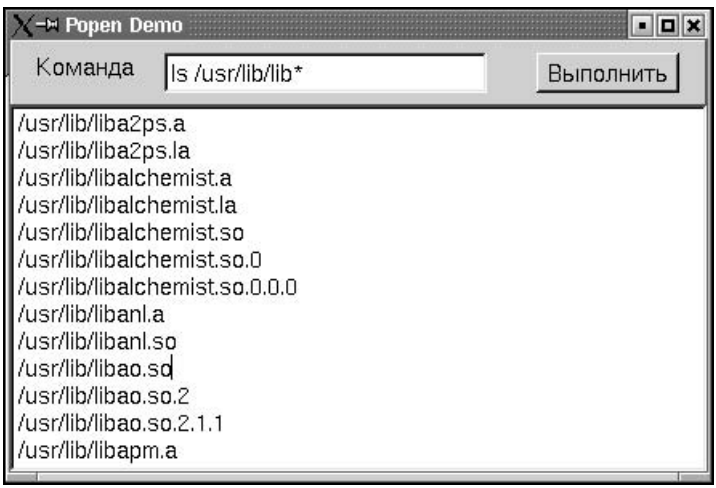


Рис. 1.5. Форма приложения PopenDemo

Команда вводится в строке компонента `EditCommand`. Затем, после нажатия кнопки `ButtonExec` или клавиши [`Enter`] (*см. обработчик* `EditCommandKeyDown`), программа запускает команду с помощью функции `popen`. При этом функция `popen` возвращает дескриптор, позволяющий нашей программе считывать данные из потока вывода запущенной команды. Особенность функции `popen` заключается в том, что эта функция возвращает корректный дескриптор файла, даже если переданная ей команда не является корректной. Самый простой способ обнаружить ошибку в этой ситуации — попытаться прочесть данные из потока вывода. Если в потоке вывода нет данных (`fread` возвращает значение 0), значит произошла ошибка.

Сокеты в файловом пространстве имен

Чаще всего сокеты упоминаются в связи с сетевым программированием, однако с их помощью можно обмениваться данными и в рамках локальной системы. Для организации взаимодействия локальных приложений при помощи сокетов в ОС Linux реализована концепция файлового пространства имен (file namespace). В рамках этой концепции сокеты могут использовать

в качестве адресов имена файлов специального типа. Важной особенностью этих сокетов является то, что соединение с их помощью локального и удаленного приложений невозможно, даже если файловая система, в которой создан сокет, доступна удаленной операционной системе.

Для работы с адресами из файлового пространства используется специальная структура данных `sockaddr_un`, определенная в файле sys/un.h. В листинге 1.19 приводится фрагмент программы, в котором сокет связывается с файлом.

Листинг 1.19. Пример использования сокета (серверная часть)

```
#include <sys/socket.h>
#include <sys/un.h>
...
struct sockaddr_un name;
int sock, size;
char * fn = "/tmp/filesocket";
sock = socket(PF_FILE, SOCK_DGRAM, 0);
name.sun_family = AF_FILE;
strcpy(name.sun_path, fn);
size = offsetof(struct sockaddr_un, sun_path) + strlen(name.sun_path) +
1;
bind(sock, (struct sockaddr *) &name, size);
...
```

В результате выполнения приведенного фрагмента на диске будет создан файл /tmp/filesocket. По окончании работы с сокетом этот файл должен быть удален. Значение, возвращенное функцией `socket`, является файловым дескриптором и может быть использовано, например, для чтения данных:

```
act_size = read (sock, buf, buf_size);
```

Приложение-клиент может соединиться с уже созданным сокетом при помощи функции `connect` (листинг 1.20).

Листинг 1.20. Пример использования сокета (клиентская часть)

```
#include <sys/socket.h>
#include <sys/un.h>
...
struct sockaddr_un name;
int sock;
```

```
char * fn = "/tmp/filesocket";
char * greeting = "How're you?";
sock = socket(PF_FILE, SOCK_DGRAM, 0);
name.sun_family = AF_FILE;
strcpy(name.sun_path, fn);
connect(sock, (struct sockaddr *) &name, sizeof(name));
write(sock, greeting, strlen(greeting));
...
```

Связанные сокеты

Связанные сокеты (socket pairs) похожи на анонимные однонаправленные каналы. Как и анонимные каналы, связанные сокеты предназначены для обмена данными между родительским и дочерним процессами, однако в отличие от каналов связанные сокеты позволяют передавать данные в обоих направлениях. Связанные сокеты создаются при помощи функции `socketpair`. Как и функция `pipe`, функция `socketpair` возвращает два файловых дескриптора, но в отличие от `pipe` оба эти дескриптора открыты и для записи, и для чтения. Рассмотрим программу sockpair.cpp, создающую дочерний процесс и реализующую двунаправленный обмен данными при помощи связанных сокетов (листинг 1.21).

Листинг 1.21. Программа sockpair.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <strings.h>
#define BUF_SIZE 512
int main ()
{
  int sockets[2];
  int size, pid;
  char buf[BUF_SIZE];
/* Создаем пару сокетов */
socketpair(AF_FILE, SOCK_DGRAM, 0, sockets)
/*  Создаем дочерний процесс */
  pid = fork();
```

```
  if (pid)
  {
  /* Родительский процесс */
    char * response = "Спасибо, хорошо!";
    size = read(sockets[1], buf, BUF_SIZE);
    printf("Дочерний процесс спросил: %s\n", buf);
    write(sockets[1], response, strlen(response) + 1);
    close(sockets[1]);
  }
  else
  {
    /* дочерний процесс */
    char * greeting = "Привет, как дела?";
    write(sockets[0], greeting, strlen(greeting) + 1);
    size = read(sockets[0], buf, BUF_SIZE);
    if (size = BUF_SIZE) buf[BUF_SIZE - 1] = '\0';
    printf("Родительский процесс ответил: %s\n", buf);
    close(sockets[0]);
  }
  return 1;
}
```

Сообщения

Сообщения Linux не связаны с графической оболочкой системы, как сообщения Windows. В ОС Linux любые процессы могут обмениваться сообщениями. Перечислим основные сходства и отличия системы сообщений в двух ОС.

- Как и сообщения Windows, сообщения Linux могут накапливаться в очереди.
- В ОС Linux в отличие от ОС Windows нет системной очереди сообщений. Каждая группа процессов, использующая обмен сообщениями, создает собственную очередь.
- В ОС Linux возможен произвольный доступ к сообщениям, находящимся в очереди, на основе их идентификаторов.
- В отличие от сообщений Windows сообщения Linux могут иметь произвольную структуру и размер.

Прототип структуры сообщения Linux приводится в листинге 1.22.

Листинг 1.22. Структура сообщения ОС Linux

```
struct msgp  {
  long mtype;
  char mtext[SOMEVALUE];
};
```

Фактически поле `mtype` является единственным обязательным полем в структуре сообщения. В этом поле хранится произвольный идентификатор сообщения. Кроме поля `mtype`, структура данных сообщения может содержать любое количество других полей различных типов.

Все типы, константы и функции, использующиеся при работе с сообщениями, объявлены в файлах sys/ipc.h и sys/msg.h (и модуле `Libc`). Очередь сообщений создается при помощи функции `msgget`. Первый параметр этой функции — ключ — это уникальное, в рамках системы, число, идентифицирующее очередь. Для получения идентификаторов можно использовать функцию `ftok`, однако можно выбрать ключ и самостоятельно, взяв для этого какое-нибудь "случайное" число. Конечно, существует определенная вероятность того, что выбранный ключ не будет уникальным, но и функция `ftok` также не может на сто процентов гарантировать уникальность полученного с ее помощью ключа. Второй параметр функции `msgget` представляет собой маску прав доступа к создаваемой очереди (аналогичную маске прав доступа к файлам) и несколько дополнительных флагов. Флаг `IPC_CREATE` указывает, что в результате вызова `msgget` должна быть создана новая очередь. При установке флага `IPC_EXCL` функция `msgget` вернет сообщение об ошибке, если очередь с указанным ключом уже существует.

Передача и получение сообщений выполняется при помощи функций `msgsnd` и `msgrcv` соответственно. Первым параметром обеих функций является идентификатор очереди, возвращенный функцией `msgget`. Во втором параметре передается размер структуры сообщения. Третий параметр функции `msgrcv` позволяет идентифицировать сообщение, которое функция желает извлечь из очереди. Идентификация сообщений основана на значении поля `mtype` структуры сообщения. Если значение третьего параметра `msgrcv` больше нуля, из очереди будет извлечено сообщение с соответствующим значением идентификатора. Если этот параметр равен нулю, из очереди будет извлечено любое первое сообщение, а если параметр отрицательный, из очереди будет извлечено первое сообщение, чей идентификатор меньше либо равен абсолютному значению параметра. Программа, читающая сообщения из очереди, не может получить данные о размере и структуре сообщения непосредственно и должна определять размер сообщения на основе его идентификатора по предварительной "договоренности" между посылающим и принимающим процессами.

Последний параметр в обеих функциях позволяет задать дополнительные флаги. Обычно функция `msgrcv` приостанавливает выполнение программы

до тех пор, пока в очереди не появится ожидаемое сообщение. При указании флага `IPC_NOWAIT` функция вернет сообщение об ошибке, если операция извлечения сообщения не может быть выполнена немедленно.

Функция `msgctl` позволяет управлять очередью и получать данные о ее состоянии. Например, вызов

```
msgctl(msgid, IPC_RMID, 0);
```

где `msgid` — идентификатор очереди сообщений, возвращенный функцией `msgget`, удаляет ранее созданную очередь.

Следует отметить, что в процессе обмена сообщениями с помощью данной очереди может участвовать более двух процессов, и все они могут как посылать, так и принимать сообщения.

В качестве примера использования сообщений Linux напишем Delphi Language-модуль содержащий функции для передачи сообщений между процессами (листинг 1.23).

Листинг 1.23. Модуль `Messages`

```
unit Messages;

interface

uses
  Libc;

type
  // Базовая структура сообщения
  TMsgStruct = record
    Size : Integer;
  end;
  // Значения, возвращаемые функциями
  TMsgRes = (mrOK,       // Вызов успешен (все функции)
             mrFailure,       // Ошибка (все функции)
             mrQueueExists,   // Очередь уже существует (CreateQueue)
             mrNotOwner,// Очередь создана другим процессом (DeleteQueue)
             mrNoMessage      // Нет сообщений в очереди (GetMessage)
            );
  // Создать очередь
  function CreateQueue(key : Integer) : TMsgRes;
```

```
  // Удалить очередь
  function DeleteQueue : TMsgRes;
  // Отправить сообщение
  function SendMessage(var Msg) : TMsgRes;
  // Принять сообщение (асинхронный режим)
  function GetMessage(Size : Integer; var Msg) : TMsgRes;
  // Принять сообщение (блокирующий режим)
  function WaitMessage(Size : Integer; var Msg) : TMsgRes;

implementation
var
  MSGId : Integer;
  Owner : Boolean = True;

  function CreateQueue;
  begin
    // $1B6 =  rw-rw-rw-
    MSGId := msgget(key, IPC_CREAT or IPC_EXCL or $1B6);
    if MSGId = -1 then
    begin
      Result := mrFailure;
      if errno() = EEXIST then
      begin
        MSGId := msgget(key, IPC_CREAT or $1B6);
        if MSGId <> -1 then
        begin
          Owner := False;
          Result := mrQueueExists;
        end;
      end;
      Exit;
    end;
    Result := mrOK;
  end;

  function DeleteQueue;
  begin
    if not Owner then
```

```
    begin
      Result := mrNotOwner;
      Exit;
    end;
    if msgctl(MSGId, IPC_RMID, nil) = -1 then
    Result := mrFailure
    else Result := mrOK;
  end;

  function SendMessage;
  var
    M : TMsgStruct;
  begin
    M := TMsgStruct(Msg);
    if msgsnd(MSGId, Msg, M.Size, 0) = -1 then
    Result := mrFailure
    else Result := mrOK;
  end;

  function GetMessage;
  begin
    if msgrcv(MSGId, Msg, Size, Size, IPC_NOWAIT) = -1 then
    begin
      if errno() = ENOMSG then
      Result := mrNoMessage
      else Result := mrFailure;
    end
    else Result := mrOK;
  end;

  function WaitMessage;
  begin
    if msgrcv(MSGId, Msg, Size, Size, 0) = -1 then
    Result := mrFailure
    else Result := mrOK;
  end;
end.
```

Обратим внимание на то, что в модуле `Messages` мы используем нетипизированную переменную `Msg`. Использование нетипизированных переменных в этом контексте очень удобно, так как функции модуля `Messages` служат для передачи сообщений произвольного размера и структуры. При использовании нашего модуля очередь сообщений должна удаляться тем же процессом, который ее создал. В качестве идентификатора сообщения используется его размер. Функция `GetMessage` не блокирует процесс. Если в момент ее вызова в очереди не было сообщений указанного размера, функция возвращает значение `mrNoMessage`. Функция `WaitMessage` блокирует процесс до тех пор, пока в очереди не появится сообщение указанного размера, либо пока очередь не будет удалена.

Листинг 1.24 демонстрирует отправку и прием сообщения с помощью модуля `Messages`.

Листинг 1.24. Отправка и прием сообщения

```
type
  TMyMsg = record
     Size : Integer;
     ... // Поля для данных сообщения
  end;
var
  Msg : TMyMsg;
...
  Msg.Size := SizeOf(Msg);
  SendMessage(Msg);
...
  Size := SizeOf(Msg);
  if GetMessage(Size, Msg) = mrOK then ...
```

Во избежание ошибки сегментации переменную `Msg` следует объявлять как глобальную.

Разделяемая память

Использование разделяемой памяти в Linux очень похоже на использование разделяемой памяти в Windows. Разделяемые блоки памяти представляют собой область памяти, отображающую адресное пространство нескольких процессов. Функции для работы с разделяемой памятью объявлены в файлах sys/ipc.h sys/shm.h и, конечно, в модуле `Libc`.

Разделяемый блок памяти выделяется при помощи функции `shmget`, которой передаются три параметра. В первом параметре передается ключ, иден-

тифицирующий выделяемый блок памяти. Смысл этого параметра аналогичен смыслу ключа в системе передачи сообщений. Второй параметр позволяет указать размер блока памяти (в байтах). В третьем параметре передается маска прав доступа и флаги, аналогичные флагам `msgget`. Функция `shmget` возвращает идентификатор выделенного блока памяти (его не следует путать с указателем на блок). Отображение блока памяти в адресное пространство процесса выполняет функция `shmat`. У функции `shmat` три параметра. Первый параметр — это идентификатор, возвращенный функцией `shmget`. Во втором параметре передается желательный адрес начала области отображения блока. Функция `shmat` "постарается" отобразить блок в указанную область, хотя успешный результат не гарантирован. Если в этом параметре передать значение `NULL`, функция сама выберет начальный адрес области отображения. Третий параметр позволяет задать дополнительные флаги. Например, флаг `SHM_RDONLY` присваивает отображаемой области статус "только для чтения". При успешном выполнении функция `shmat` возвращает указатель на начало области отображения. Функция `shmdt` удаляет область отображения (но не блок разделяемой памяти). Единственный параметр этой функции — указатель, возвращенный функцией `shmat`. Для удаления блока разделяемой памяти нужно вызвать функцию `shmctl`:

```
shmctl(shmid, IPC_RMID, 0);
```

где `shmid` — идентификатор, возвращенный функцией `shmget`.

После получения указателя на область разделяемой памяти с ней можно работать точно так же, как и с любой локальной областью памяти программы. При этом вы должны помнить, что содержимое области разделяемой памяти может быть изменено несколькими независимыми процессами. Следует также учесть, что в разных процессах блок разделяемой памяти скорее всего, будет иметь разные базовые адреса, поэтому при работе с разделяемой памятью вся адресация должна быть относительной и передача непосредственных указателей не допускается.

Механизм разделяемой памяти ОС Linux можно использовать для решения самых разных задач. Мы используем этот механизм при написании функций, ограничивающих число одновременно запущенных экземпляров программы. Часто бывает необходимо, чтобы в данный момент времени в системе выполнялось не более одной копии приложения. Такое ограничение может потребоваться, если приложение контролирует доступ к критическим ресурсам, либо в случае многооконной среды, когда вместо запуска нового экземпляра приложения лучше открыть еще одно окно. Решение этой задачи с помощью разделяемой памяти выглядит следующим образом: при запуске процесс создает блок разделяемой памяти с определенным ключом и записывает в этот блок некоторое "магическое" значение. Предварительно процесс проверяет, не был ли этот блок уже инициализирован подобным образом. Если да — значит другая копия процесса уже запущена, если

нет — данный экземпляр процесса является единственным в системе. В листинге 1.25 приводится текст файла oneinst.cpp, в котором описаны функции `LockInst` и `UnlockInst`.

Листинг 1.25. Файл oneinst.cpp

```
//---------------------------------------------------------------------------

#include <SysUtils.hpp>
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#pragma hdrstop

#include "oneinst.h"
//---------------------------------------------------------------------------

static int shmid;

bool LockInst(key_t key)
{
  AnsiString S;
  struct memarea
  {
    key_t magic;
    pid_t pid;
  } * pmemarea;
  shmid = shmget(key, sizeof(struct memarea), IPC_CREAT | 0x1B6);
  pmemarea = (struct memarea*) shmat(shmid, NULL, 0);
  if (pmemarea->magic != key)
  {
    pmemarea->magic = key;
    pmemarea->pid = getpid();
    shmdt((void*) pmemarea);
    return true;
  }
  S = Format("/proc/%d", ARRAYOFCONST((pmemarea->pid)));
  if (FileExists(S))
```

```
  {
    shmid = 0;
    shmdt((void*) pmemarea);
    return false;
  }
  else
  {
    pmemarea->magic = key;
    pmemarea->pid = getpid();
    shmdt((void*) pmemarea);
    return true;
  }
}

bool UnlockInst()
{
  if (shmid != 0)
  {
    if (shmctl(shmid, IPC_RMID, NULL) == 0) return true;
    return false;
  }
  return true;
}
```

Функция `LockInst` проверяет, не запущена ли уже копия данной программы. Функции передается значение-ключ для блока разделяемой памяти. Если других копий программы в памяти нет, функция `LockInst` создает "заметку", записывая в блок разделяемой памяти "магическое" число (для этой цели используется значение ключа), и возвращает значение `true`. Если "заметка" уже существует, функция возвращает значение `false`.

Функция `UnlockInst` удаляет "заметку", созданную функцией `LockInst`. Эту функцию нужно вызывать в блоке завершения работы программы. На первый взгляд код функции `LockInst` может показаться излишне сложным. Не достаточно ли проверить наличие блока разделяемой памяти с записанным в него "магическим" числом? Проблема заключается в том, что объект разделяемой памяти может существовать, даже если создавший его процесс уже завершился. Объекты межпроцессного взаимодействия не удаляются автоматически по завершении создавшего их процесса. Их всегда нужно удалять явно. Это значит, что если процесс, создавший такой объект, завершился некорректно, например, в результате нажатия комбинации клавиш <Ctrl>+<C>, созданный им объект может остаться в системе. В нашем слу-

чае это означает, что после некорректного (без вызова `UnlockInst`) завершения одной копии программы мы не смогли бы запустить ее еще раз до перезагрузки системы. Для того чтобы функция `LockInst` работала корректно, она должна проверять не только наличие инициализированного блока разделяемой памяти в системе, но и то, существует ли соответствующий процесс. Для этого в область разделяемой памяти записывается не только "магическое" число, но и идентификатор процесса-владельца. С помощью файловой системы `/proc` функция `LockInst` проверяет, существует ли еще процесс с указанным идентификатором. Если такого процесса не существует, функция делает текущий процесс владельцем блока разделяемой памяти (маска разрешений это позволяет). Переменная `shmid`, хранящая идентификатор объекта разделяемой памяти, выполняет также роль флага, благодаря которому функция `UnlockInst` не удаляет объект, если текущий процесс не является его владельцем.

Семафоры

Семафоры широко используются в системе Linux как средство синхронизации потоков, принадлежащих одному или разным процессам, а также для разделения доступа к критическим ресурсам.

Состояние семафора определяется значением некоторой внутренней переменной. Функция, обращающаяся к семафору, либо изменяет значение этой переменной, либо приостанавливает выполнение вызвавшего ее потока до тех пор, пока это значение не будет изменено другим потоком.

Работу семафоров можно продемонстрировать на примере разделения доступа к некоему ресурсу, не допускающему одновременного использования несколькими потоками. Перед тем, как начать работу с таким ресурсом, поток должен проверить, не используется ли ресурс другим потоком. Если ресурс занят, поток ожидает его высвобождения. Если ресурс свободен, поток начинает работу с ресурсом. При этом поток сообщает системе, что ресурс занят, и другие потоки не могут получить к нему доступ. По окончании работы с ресурсом поток сигнализирует системе о высвобождении ресурса.

Семафоры создаются при помощи функции `semget`, объявленной в файле sem.h. У функции `semget` три параметра. В первом параметре функции передается уникальный ключ, аналогичный по смыслу соответствующему параметру функций `msgget` и `shmget`. Второй параметр указывает, сколько семафоров необходимо создать. Дело в том, что часто для синхронизации процессов требуется более одного семафора, и функция `semget` позволяет создавать несколько семафоров за один вызов.

Для управления состоянием семафоров служит функция `semop`. Первый параметр этой функции — идентификатор, возвращенный функцией `semget`. Вто-

рой параметр функции semop представляет собой указатель на массив структур, содержащих данные об операции над семафорами. В третьем параметре передается число записей в массиве, которое должно соответствовать числу семафоров. Структура, служащая для передачи в функцию semop данных об операциях над семафорами, называется sembuf. И структура и функция объявляются в файле sys/sem.h. В структуре sembuf определено много полей, из которых явным образом обязательно должны быть заданы три:

- short sem_num — номер семафора, над которым выполняется операция (нумерация начинается с нуля);
- short sem_op — число, изменяющее состояние семафора;
- short sem_flg — дополнительные флаги.

Состояние семафора определяется значением некоторой внутренней переменной. Это значение может быть изменено при помощи поля sem_op. Следующая функция (листинг 1.26) иллюстрирует логику работы функции semop в зависимости от значения переменной состояния (semvalue) и переменной sem_op.

Листинг 1.26. Принцип работы семафора

```
void semaphore (int semvalue, int sem_op)  {
  if (sem_op != 0)  {
    if (sem_op < 0)  while (semvalue < ABS(sem_op));
    semvalue += sem_op;
  }
  else while (semvalue != 0);
}
```

Отрицательное значение sem_op соответствует операции проверки доступности ресурса и вызывает приостановку потока, если ресурс недоступен. Положительное значение сигнализирует о высвобождении ресурса.

В поле sem_flg устанавливаются флаги для функции semop. Обязательным является флаг SEM_UNDO. Кроме этого можно указать флаг IPC_NOWAIT. В этом случае при наступлении "блокирующей ситуации" функция semop не приостанавливает выполнение программы, а возвращает значение −1.

Для управления семафорами используется функция semctl, аналогичная shmctl. У этой функции, однако, есть дополнительные возможности. Например, вызов

```
semctl(semid, semnum, SETVAL, value);
```

позволяет установить значение переменной, управляющей состоянием семафора, равным value. Рассмотрим простой пример (листинг 1.27).

Листинг 1.27. Пример использования семафоров

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#define KEY 1421
int main()
{
  int pid, semid;
  struct sembuf buf;
  semid = semget(KEY, 1, IPC_CREAT|IPC_EXCL|0x1B6);
  buf.sem_num = 0;
  buf.sem_flg = SEM_UNDO;
  semctl(semid, 0, SETVAL, 0);
  pid = fork();
  if (pid == 0) {
    sleep(1);
    printf("Здравствуй,\n");
    buf.sem_op = 1;
    semop(semid, (struct sembuf*) &buf, 1);
    buf.sem_op = -2;
    semop(semid, (struct sembuf*) &buf, 1);
  } else {
    buf.sem_op = -1;
    semop(semid, (struct sembuf*) &buf, 1);
    printf("Мир!\n");
    buf.sem_op = 2;
    semop(semid, (struct sembuf*) &buf, 1);
    semctl(semid, 0, IPC_RMID, 0);
  }
  return 0;
}
```

Сначала мы создаем два процесса. Родительский процесс печатает слово "Мир!" только после того, как дочерний процесс напечатает "Здравствуй,". Вызов `sleep(1)` приводится для наглядности. Второй вызов функции `semop` с параметром `sem_op`, равным 2, производится для того, чтобы дочерний процесс не завершился до окончания родительского (при работе с семафорами следует уделять внимание синхронизации завершения работы).

В заключение этого раздела стоит указать две консольные команды ОС Linux, служащие для управления объектами межпроцессного взаимодействия: `ipcs` и `ipcrm`. Команда `ipcs` выводит данные о наличествующих в данный момент очередях сообщений, блоках разделяемой памяти и семафорах. Команда `ipcrm` позволяет удалять эти объекты по заданному идентификатору. Данная команда может быть полезна в процессе отладки IPC-приложений при появлении "бесхозных" межпроцессных объектов.

Разделяемые библиотеки и объектные файлы

В разделе, посвященном консольным приложениям, мы уже познакомились с библиотеками ОС Linux. Этот раздел посвящен вопросам эффективного взаимодействия Kylix-приложений с библиотеками, а также созданию библиотек в среде Kylix 3.

Как уже отмечалось, библиотеки ОС Linux делятся на *статические* и *динамические*. При всей похожести динамических библиотек ОС Linux на библиотеки ОС Windows DLL между ними существуют определенные различия. Первое отличие, с которым Windows-программисту придется столкнуться при работе с динамическими библиотеками ОС Linux, заключается в наличии у библиотеки нескольких имен. У каждой библиотеки есть внутреннее имя (soname), записанное в теле самой библиотеки и служащее для идентификации библиотеки загрузчиком. Внутреннее имя библиотеки может отличаться от имени файла, в котором она хранится. Общая структура имени динамической библиотеки может быть представлена следующей схемой:

```
lib<имя_библиотеки>.so.<версия>
```

где `<версия>` — номер версии библиотеки, состоящий из нескольких цифр, разделенных точками. Внутреннее имя библиотеки зачастую отличается от имени файла библиотеки только более укороченным номером версии. Например, библиотека с внутренним именем

```
libmylibrary.so.1
```

может храниться в файле с именем

```
libmylibrary.so.1.0.2
```

Кроме этих имен, для облегчения связывания библиотек с приложением создаются символические ссылки, упрощающие имя библиотеки. Например, если библиотека `libmylibrary.so.1.0.2` размещена в каталоге /usr/lib, на нее могут указывать ссылки

```
/usr/lib/libmylibrary.so.1 -> libmylibrary.so.1.0.2
/usr/lib/libmylibrary.so -> /usr/lib/libmylibrary.so.1
```

Заметим, что имя одной из ссылок совпадает с внутренним именем библиотеки.

Тут возникает одна тонкость. В процессе статического связывания программы с динамическими библиотеками компоновщику обычно указывается самое "короткое" имя библиотеки, в нашем случае — `libmylibrary.so`. В Kylix 3 это можно сделать при помощи ключевого слова **external** в Delphi Language, директивы `#pragma link` или в настройках проекта в C++ *(см. разд. "Особенности C++ в Borland Kylix" данной главы)*. Однако следует учитывать, что фактически приложение будет связано с файлом libmylibrary.so.1. Имя этого файла соответствует внутреннему имени библиотеки и, как видим, содержит номер версии. Если в будущем дистрибутиве версия библиотеки изменится, приложение уже не сможет найти ее. Для решения проблемы несоответствия номеров версий можно создать еще одну символическую ссылку на файл библиотеки с именем, соответствующим прежней версии. Вот почему на некоторые библиотеки ОС Linux в каталоге /usr/lib дается так много разных ссылок.

Кроме статического связывания динамических библиотек в ОС Linux существует возможность их динамического подключения.

Примечание

Эти термины могут вызвать путаницу: статическое подключение динамических библиотек выполняется на этапе компоновки. Это не значит, что библиотека становится частью приложения, но в программном модуле сохраняется фиксированная ссылка на библиотеку, которая должна быть разрешена в процессе загрузки модуля на исполнение. В отличие от этого при динамическом подключении библиотек их загрузка выполняется при помощи специальных функций во время выполнения программы.

Динамическое подключение библиотек ОС Linux очень похоже на динамическое подключение ОС Windows DLL. Для работы с динамически загружаемыми библиотеками в ОС Linux служат функции:

- `dlopen` — отобразить указанный файл динамической библиотеки в адресное пространство процесса;
- `dlclose` — удалить созданное отображение;
- `dlsym` — получить адрес объекта загруженной библиотеки по его имени;
- `dlerror` — получить информацию об ошибках, возникших при вызове функций динамически загружаемых библиотек.

Программисты, разрабатывающие приложения на языке C++, найдут прототипы этих функций в файле dlfcn.h, а пользователи Delphi Language — в модуле `Libc` (эти функции являются частью `glibc`).

В листинге 1.28 представлен фрагмент модуля, динамически загружающего функции `GLX` из библиотеки libGL.so.

Листинг 1.28. Фрагмент модуля, динамически импортирующего функции GLX

```
interface
...
const
 LibName = '/usr/lib/libGL.so';

var
  glXChooseVisual: function(dpy: PDisplay; screen: Integer;
                            attribList: PInteger): PXVisualInfo; cdecl;
  glXCopyContext: procedure(dpy: PDisplay; src, dst: GLXContext;
                            mask: LongWord); cdecl;
  glXCreateContext: function(dpy: PDisplay; vis: PXVisualInfo;
                             shareList: GLXContext;
direct: Boolean): GLXContext; cdecl;
...
implementation
var
   LibHandle : Pointer = nil;

initialization
 LibHandle := dlopen(LibName, RTDL_NOW or RTDL_GLOBAL);
 if LibHandle <> nil then
 begin
    glXChooseVisual := dlsym(LibHandle, 'glXChooseVisual');
    glXCopyContext := dlsym(LibHandle, 'glXCopyContext');
    glXCreateContext := dlsym(LibHandle, 'glXCreateContext');
   ...
end else raise Exception.Create(Format('Не удалось загрузить ↳
библиотеку %s', [LibName]);

finalization
  if LibHandle <> nil then dlclose(LibHandle);
end.
```

Для импортирования функций из библиотеки нам потребуются переменные процедурного типа (или указатели на функции в случае C++).

Первый параметр функции `dlopen` — имя загружаемой библиотеки. Второй параметр позволяет указать флаги, влияющие на процесс загрузки. Флаг `RTDL_NOW` указывает, что отображение должно создаваться в момент вызова

dlopen. Если вместо этого флага указать RTLD_LAZY, отображение будет загружено только тогда, когда в нем возникнет необходимость, например, при получении указателя на объект, экспортируемый библиотекой. Флаг RTDL_GLOBAL указывает, что все символы, экспортируемые библиотекой, должны быть видимы в контексте загружающего процесса. Использование этого флага необходимо в некоторых ситуациях, о которых будет сказано ниже. Если библиотека была загружена успешно, функция dlopen возвращает указатель, не равный nil. Этот указатель будет использоваться при обращении к другим функциям загрузки библиотек.

Адрес объекта, экспортируемого библиотекой, возвращает функция dlsym. Первый параметр функции — указатель, возвращенный dlopen, второй параметр — строка, содержащая имя объекта. Если объект с указанным именем не был найден в библиотеке, функция dlsym возвращает значение nil.

Как и в случае ОС Windows, динамическое подключение в ОС Linux замедляет процесс загрузки программы. Однако в ОС Linux у динамического подключения библиотек есть ряд преимуществ, которые делают этот способ подключения предпочтительным.

Во-первых, загрузка библиотек во время выполнения программы решает проблему изменения имен. Отметим то, что в приведенном примере мы использовали "короткое" имя библиотеки, которое не меняется вместе с версиями. Во-вторых, решается проблема размещения библиотек. Как правило, при статическом связывании приложение предполагает найти библиотеку лишь в определенных каталогах, тех, которые сконфигурированы для поиска библиотек по умолчанию. Однако нет никакой гарантии, что в каждом дистрибутиве ОС Linux искомые библиотеки будут размещены именно в этих каталогах. Загружая библиотеки во время выполнения программы, можно организовать "интеллектуальный" поиск файла библиотеки или предоставить пользователю возможность указывать местонахождение библиотек в файлах конфигурации приложения.

Есть еще две проблемы, характерные для Delphi language и решаемые с помощью динамической загрузки библиотек. Первая проблема связана с видимостью имен. Некоторые библиотеки, используемые в комплексе, ссылаются друг на друга, не будучи связаны между собой на уровне компоновщика. Такая ситуация возникает, например, с библиотеками libvorbis.so и libvorbisfile.so. Библиотека libvorbisfile.so использует функции из библиотеки libvorbis.so, хотя явно с этой библиотекой не связана. Проверить, с какими библиотеками связан тот или иной исполнимый модуль, можно при помощи команды ldd. В листинге 1.29 приводится результат выполнения команды

```
ldd /usr/lib/libvorbisfile.so
```

Листинг 1.29. Результат выполнения команды `ldd` для библиотеки libvorbisfile.so

```
libm.so.6 => /lib/libm.so.6 (0x40017000)
libogg.so.0 => /usr/lib/libogg.so.0 (0x40039000)
libc.so.6 => /lib/libc.so.6 (0x4003e000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

Как видим, библиотеки libvorbis.so в этом списке нет. Для приложений, написанных на C/C++, это не создает никаких проблем, так как во время выполнения приложения все имена из всех статически связанных динамических библиотек будут видимы, а вот статическое связывание таких библиотек в среде Delphi Language приведет к ошибкам. Для решения этой проблемы соответствующие библиотеки следует загружать динамически с указанием флага `RTLD_GLOBAL`.

Вторая проблема во "взаимоотношениях" между программами на Delphi Language и динамическими библиотеками связана с экспортированием переменных. В ОС Linux библиотеки могут экспортировать не только функции, но и переменные. К сожалению, в Delphi Language нет механизма непосредственного обращения к библиотечным переменным. Но и тут динамическая загрузка библиотек может решить проблему. Дело в том, что функция `dlsym` позволяет получить адрес любого объекта, экспортируемого библиотекой, в том числе и переменной. Предположим, что библиотека `libmylibrary.so` экспортирует переменную `intval` типа `int`. Импортировать эту переменную в Delphi-модуле можно следующим образом (листинг 1.30).

Листинг 1.30. Импорт переменной из динамической библиотеки для Delphi Language

```
var
  intval_p : PInteger;
 ...
 intval_p := dlsym(LibraryHandle, 'intval');
```

Таким образом, динамическая загрузка библиотек является чрезвычайно мощным и гибким средством, и неудивительно, что Kylix 3 широко использует этот механизм для подключения модулей времени выполнения.

Процесс создания динамических библиотек в средах Kylix мало чем отличается от создания Windows DLL в средствах разработки от Borland для Windows. Однако, некоторые отличия все же есть. Если мы создадим библиотеку общего назначения на Delphi Language, требуется указать формат вызова `cdecl` для экспортируемых функций. Как уже отмечалось, при компоновке библиотеки следует указать ее внутреннее имя. Для этого в Delphi Language введены специальные директивы компилятора (табл. 1.7).

Таблица 1.7. *Директивы компилятора Delphi Language для динамических библиотек*

Директива	Объяснение	Пример
`$SONAME`	Внутреннее имя библиотеки	`{$SONAME 'libmylib.so.1'}`
`$SOVERSION`	Версия библиотеки	`{$SOVERSION '1.0.2'}`
`$SOPREFIX`	Префикс имени библиотеки	`{$SOPREFIX 'lib'}`
`$SOSUFFIX`	Суффикс имени библиотеки	`{$SOSUFFIX '1.2.3'}`

У директив `$SONAME` и `$SOPREFIX` есть значение по умолчанию. Для `$SONAME` значение по умолчанию совпадает с именем проекта с добавлением префикса `'lib'` и расширения `'so'`. Значение по умолчанию директивы `$SOPREFIX` равно `'lib'`. Значения всех этих параметров можно указать также в настройках проекта. В C++ соответствующие опции компилятора не определены, и параметры библиотеки можно указать только в настройках проекта (рис. 1.6).

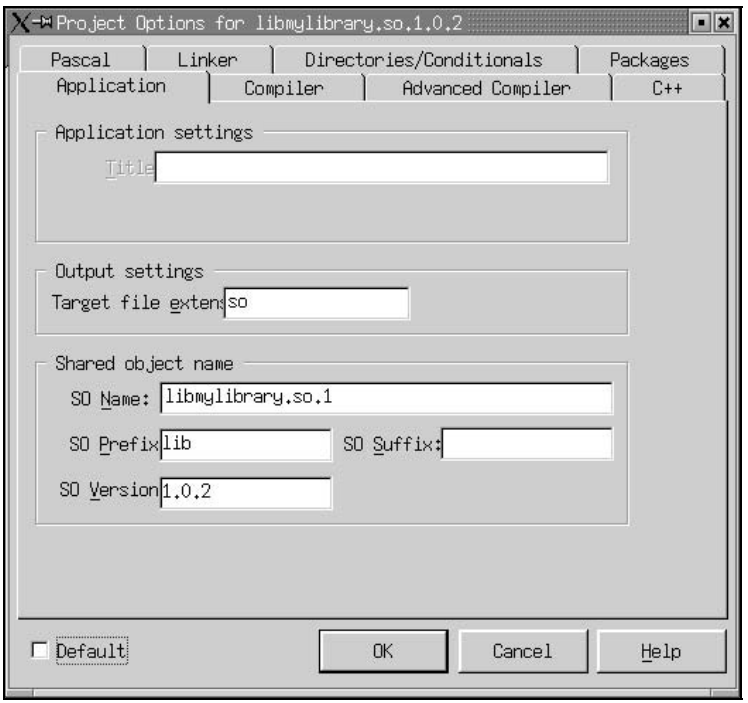


Рис. 1.6. Окно настроек проекта динамической библиотеки в Kylix C++

Разрабатывая библиотеки на Kylix C++ для использования в ОС Linux, следует помнить о несовместимости форматов классов Kylix C++ и GNU C++. Библиотека, написанная на Kylix C++, будет совместима только с приложениями, созданными в Kylix C++ IDE. Если нужно создать с помощью Kylix C++ IDE библиотеку, которую можно было бы использовать в приложениях, написанных на GNU C/C++ или Delphi Language, придется либо выбрать в качестве языка библиотеки стандартный C (Kylix C++ IDE это позволяет), либо экспортировать функции библиотеки в формате C. Кстати, последнее сделать совсем не трудно. Вернемся к модулю `oneinst` из предыдущего раздела (листинг 1.25). Из текста модуля видно, что в нем используется заголовочный файл oneinst.h, исходный текст которого до сих пор не приводился. Очевидно, что файл oneinst.h должен содержать объявление функций `LockInst` и `UnlockInst`. В листинге 1.31 приводится вариант файла oneinst.h для экспорта указанных функций в формате C.

Листинг 1.31. Заголовочный файл модуля oneinst

```
#include <sys/types.h>

#ifndef ONEINST_H
#define ONEINST_H

#ifdef __cplusplus
extern "C" {
#endif

bool LockInst(key_t key);
bool UnlockInst();

#ifdef __cplusplus
}
#endif

#endif
```

Заметим, что благодаря директиве условной компиляции `#ifdef __cplusplus` заголовочный файл oneinst.h можно использовать и при компиляции самой библиотеки, и в качестве интерфейса при подключении библиотеки к другому модулю. В том что библиотека, скомпилированная с таким заголовочным файлом, экспортирует функции с именами, отвечающими требованиям C, можно убедиться с помощью команды `nm`.

До сих пор мы рассматривали только динамические библиотеки, но Kylix C++ позволяет использовать и статические. А можно ли создать ста-

тическую библиотеку в Kylix C++? Конечно, можно. Как уже отмечалось, статические библиотеки — это фактически хранилище объектных файлов, которые компонуются с программным модулем.

Для создания статической библиотеки в Kylix C++ следует выбрать пункт **Library** в диалоговом окне **New Items**. После этого Kylix предложит включить в проект библиотеки файлы модулей. Можно включать как исходные на C/C++, так и уже скомпилированные объектные файлы. После включения всех необходимых файлов в проект даем команду `Build Project`, и новая статическая библиотека готова.

В отличие от среды Delphi Language Kylix C++ IDE сохраняет объектные файлы для каждого скомпилированного модуля. Если исходные тексты модуля были написаны на C или использовалась директива `extern "C"`, эти модули, как и созданные на их основе статические библиотеки, можно использовать совместно с компилятором GCC. Объектные файлы можно также компоновать с программами, написанными на Delphi language, как показано в листинге 1.32.

Листинг 1.32. Подключение объектного файла oneinst.o

```
{$L 'oneinst.o'}
...
function LockInst(key : Integer) : Boolean; cdecl; external;
function UnlockInst : Boolean; cdecl; external;
```

Различия между GNU C++ и Kylix C++

Создадим заготовку Kylix C++ и добавим в файл главного модуля строку:

```
#include <linux/cdrom.h>
```

Теперь попробуем скомпилировать проект.

Как ни странно, IDE выдает сообщение об ошибке в одном из заголовочных файлов, на который косвенно ссылается файл linux/cdrom.h. Несмотря на все усилия по стандартизации, версии языков C и C++ в разных компиляторах по-прежнему различаются, и язык GNU C/C++, на котором основана система Linux, не полностью совместим с Kylix C/C++. Это естественно, если учесть, что Kylix C++ прежде всего используется для применения технологий Borland на платформе ОС Linux, к тому же различия касаются лишь мелких деталей. Но дьявол, как известно, прячется в мелочах, и мелкие детали, как мы видели в приведенном выше примере, иногда приводят к большим проблемам. О несовместимости GNU C++ и Kylix C++ на уровне двоичных модулей уже говорилось в начале этой главы. В данном разделе мы рассмотрим различия на уровне синтаксиса языков.

Макросы, позволяющие отделить исходный текст, специфичный для компиляторов Borland, были рассмотрены в настоящей главе. В GCC (и G++)

также определены макросы, позволяющие идентифицировать компилятор. Наиболее часто используемые из них — `__GNUC__` (для GNU C) и `__GNUG__` (для GNU C++).

Список имен макросов, определенных по умолчанию в GCC и не определенных в Kylix C++ IDE, включает следующие имена: `__asm__`, `__const__`, `__inline__`, `__signed__`, `__volatile__`. В некоторых случаях эти макросы можно переопределить в Kylix C++ IDE, используя аналогичные макросы этой среды. Пример "доопределения" макросов приводится в листинге 1.33.

Листинг 1.33. Определение макросов GCC для Kylix IDE

```
#ifdef __BORLANDC__
#define __inline__ __inline
#define __signed__ signed
#define __const__ const
#endif
```

К часто используемым макросам GCC, не имеющим аналогов в Kylix, относятся макросы `__attribute__`, `__CHAR_UNSIGNED__`, `__OPTIMIZE__`, `__REGISTER_PREFIX__`, `__USER_LABEL_PREFIX__`.

Kylix C/C++ не поддерживает также макросредства GCC, предназначенные для работы с комплексными числами (макросы `__complex__`, `__real__` и `__imag__`).

Не поддерживаются в Kylix и такие возможности GCC, как определение макросов с переменным числом параметров, оператор `typeof`, операнды по умолчанию в операторе сравнения `"?"` (конструкции типа `x ? : y`), арифметические операции над нетипизированными указателями и указателями на функции. Kylix C/C++ не позволяет назначать переменным определенные регистры процессора, как это делается в GCC:

```
register int foo asm ("%edx");
```

и не поддерживает массивы нулевой длины:

```
char str[0];
```

Не поддерживаются также введенные в GNU C++ операторы `<?` и `>?` (определение наибольшего и наименьшего из двух значений).

Скорее всего, у читателя уже возникли вопросы: насколько важны эти различия и являются ли они непреодолимыми? Ответить на эти вопросы можно так: в большинстве ситуаций, с которыми программист, работающий в Kylix, может столкнуться на практике, эти различия не представляют непреодолимого препятствия. Если мы не собираемся компилировать в Kylix исходные тексты, специально написанные для GCC, то скорее всего столкнемся с различиями синтаксиса лишь в заголовочных файлах, а такие проблемы обычно легко решаемы. Тот же файл linux/cdrom.h совсем нетрудно адаптировать таким образом, чтобы он был "понятен" компилятору Kylix.

Глава 2

Графический интерфейс в Kylix-приложениях

В предыдущей главе мы ознакомились с общими принципами программирования в ОС Linux и методами решения некоторых распространенных задач. Хотя мы уже использовали графический интерфейс в наших проектах (в конце концов Borland Kylix — среда *визуальной* разработки), мы не рассматривали механизмы, лежащие в основе элементов графического интерфейса Kylix-приложений. Именно тонкостям программирования графического интерфейса в Kylix и посвящена данная глава.

Необходим ли программисту материал этой главы? Вполне возможно, что нет. Библиотека компонентов VisualCLX (Visual Component Library for Cross Platform, визуальный компонент кросс-платформенной библиотеки CLX) - это Kylix-аналог VCL, библиотеки визуальных компонентов Delphi Language и Borland C++ Builder, но только разработанный без использования функций Windows GUI API. На базе этой библиотеки основан графический интерфейс Kylix-приложений, и ее применение скрывает от программиста технические детали реализации графических элементов управления. Это позволяет разрабатывать пользовательский интерфейс приложений при помощи методов, хорошо знакомых программистам в среде программирования Delphi Language и Borland C++ Builder для ОС Windows. Однако стандартные компоненты позволяют решать лишь стандартные задачи. Если же нам понадобится выйти за рамки возможностей VisualCLX, а возможности эти вовсе не безграничны, мы неизбежно столкнемся с необходимостью заняться программированием графического интерфейса на низком уровне. Хотим передавать данные между нашим приложением и другими программами в графической среде ОС Linux KDE при помощи технологии Drag and Drop? Изменять внешний вид и поведение стандартных элементов управления? Отслеживать в своей программе события системы, недоступные обычному Kylix-приложению? Хотим использовать в своем приложении графические средства, отличные от тех, что предоставляет VisualCLX?

Тогда стоит прочесть эту главу.

Библиотека Qt library — основа графического интерфейса Kylix-приложений

Как отмечалось в предыдущей главе, графическая система не является неотъемлемой частью ОС Linux. Более того, сама графическая система не монолитна, а состоит из нескольких уровней, причем для большинства уровней система предоставляет несколько разных реализаций. Структура графической системы ОС Linux представлена на рис. 2.1.

Рис. 2.1. Структура графической системы Linux

Работа с графикой в ОС Linux осуществляется на основе сервера XFree86. Эта графическая подсистема предоставляет в распоряжение программиста интерфейсы для выполнения базовых операций, необходимых для построения графического интерфейса пользователя и вывода графической информации. Над сервером XFree86 существует надстройка — оконный менеджер. Эта часть системы управляет окнами — прямоугольными областями, служащими для вывода графической информации. Роль оконного менеджера в графической системе чрезвычайно важна, так как почти всякий элемент графического интерфейса приложения является в основе своей окном. Над оконным менеджером есть еще одна надстройка — *библиотека элементов управления* (Widgets). Эта библиотека определяет свойства и внешний вид основных элементов графического интерфейса. Большинство графических приложений ОС Linux взаимодействует с графической системой исключительно через библиотеку элементов управления, хотя прямой доступ к ни-

жележащим уровням и не запрещен. Практически каждый дистрибутив ОС Linux предоставляет несколько оконных менеджеров и библиотек элементов управления. Поскольку интерфейс взаимодействия между разными уровнями графической системы стандартизирован, любая библиотека элементов управления может взаимодействовать с любым оконным менеджером, если только реализация библиотеки-менеджера не содержит ошибок. Вот почему одни и те же приложения могут выполняться на самых разных оконных менеджерах.

Что же касается взаимодействия графических приложений с библиотекой элементов управления, то тут существует жесткая связь, и графическое приложение не может выполняться, если соответствующая ему библиотека не установлена в системе.

В то время, когда первая версия Kylix только разрабатывалась, среди программистов ходили упорные слухи о том, что основой графических компонентов Kylix станет *gtk* — набор графических элементов, на котором построена графическая среда ОС Linux – GNOME (GNU Network Object Model Environment, Сетевая Объектная Среда GNU). Однако разработчики Kylix сделали окончательный выбор в пользу Qt library — библиотеки, разрабатываемой норвежской компанией Trolltech. Эта библиотека лежит в основе графической среды ОС Linux – KDE (K Desktop Environment).

Библиотека Qt library является основой библиотеки VisualCLX точно так же, как Windows GUI API является основой Delphi VCL. Большинство компонентов VisualCLX можно рассматривать как "оболочки" соответствующих объектов Qt library. Однако механизм взаимодействия между VisualCLX и Qt library сложнее, чем механизм взаимодействия между VCL и Windows GUI API. Дополнительная сложность вызвана тем, что Qt library — это не набор функций, а иерархия классов C++. Как мы уже видели в предыдущей главе, в Borland Kylix невозможно импортировать классы GNU C++ непосредственно, и разработчикам Kylix пришлось пойти обходным путем. Для решения указанной проблемы между уровнями Qt library и VisualCLX был введен дополнительный слой — CLXDisplay API.

Идея CLXDisplay API заключается в трансляции методов классов GNU C++ в функции C. Для обращения к какому-либо методу экземпляра класса Qt library вызывается соответствующая функция, которой в качестве одного из параметров передается указатель на экземпляр класса, для которого должен быть вызван данный метод.

Поясним вышесказанное на простом примере: допустим, в библиотеке, скомпилированной в GNU C++, реализован класс `QSomeClass`. У этого класса есть конструктор, деструктор и метод `Method1`. Для того чтобы сделать методы класса GNU C++ доступными в Kylix, в GNU C++ компилируется библиотека-оболочка для класса `QSomeClass`, экспортирующая C-функции `QSomeClass_create`, `QSomeClass_destroy` и `QSomeClass_method1`.

О том, как экспортировать C++ функции в формате C, говорилось в предыдущей главе (*см. разд. "Разделяемые библиотеки и объектные файлы" гл. 1*).

Для того чтобы в Kylix-программе создать экземпляр класса `QSomeClass`, мы вызываем функцию `QSomeClass_create`, и передаем ей в качестве параметров значения, требующиеся конструктору класса. Написанная на GNU C++ функция `QSomeClass_create` создает экземпляр класса `QSomeClass` и передает значения своих параметров его конструктору. В качестве результата функция возвращает некий идентификатор созданного экземпляра, например, указатель на него. Таким образом, мы можем создать несколько экземпляров класса `QSomeClass`. Если теперь мы хотим вызвать метод `Method1` для одного из экземпляров `QSomeClass`, мы вызываем функцию `QSomeClass_method1`, передавая ей в качестве одного из параметров значение, возвращенное функцией `QSomeClass_create`. Оболочка класса вызывает метод `Method1` для экземпляра, соответствующего переданному идентификатору, передает этому методу необходимые параметры и затем, если нужно, возвращает значение, возвращенное методом `Method1`. Для уничтожения экземпляра `QSomeClass` вызывается функция `QSomeClass_Destroy`, которой передается идентификатор этого экземпляра.

CLXDisplay API как раз и представляет собой набор подобных C-функций для классов Qt library. В предыдущих версиях Kylix поставлялись две специальные библиотеки: *libqt* и *libqtintf*. Библиотека libqt содержала реализацию Qt library, а в библиотеке libqtintf был реализован CLXDisplay API. В новой версии Kylix обе библиотеки объединены в одну — *libborqt*. Впрочем, в целях обеспечения обратной совместимости старые библиотеки также поставляются с новой версией Kylix.

В связи с упоминанием библиотеки libborqt уместно рассказать о том, как научить приложения Kylix 3 запускаться отдельно от среды разработки. Стоит отметить, что в новой версии Kylix сделать это гораздо проще, чем в предыдущей. Для того чтобы запускать Kylix-приложения отдельно от Kylix, достаточно:

1. Разместить библиотеку libborqt в каталоге /usr/lib/.
2. В режиме root дать команды:

```
cp kylix/bin/libborqt-6.9.0-qt2.3.so /usr/lib
ln -s /usr/lib/libborqt-6.9.0-qt2.3.so \\
/usr/lib/libborqt-6.9-qt2.3.so
```

После этого программы, созданные в Kylix 3, можно запускать отдельно от Kylix.

Функции интерфейса CLXDisplay API объявляются в модуле `Qt.pas` для Delphi и файле Qt.hpp для C++. В комплект документации, поставляемой с Kylix, справочник по CLXDisplay API не входит, так что нашими главными

источниками информации по этому вопросу будут файлы Qt из поставки Kylix и документация к Qt library.

В качестве примера импортирования Qt-классов в CLXDisplay API рассмотрим Qt-класс – `QPushButton`. Класс `QPushButton` реализует кнопку — элемент пользовательского интерфейса и является основой класса `TButton` из библиотеки VisualCLX. У класса `QPushButton` есть три конструктора, деструктор и ряд методов, таких как `SetDefault`, `IsDefault`, `SetFlat` и т. д. Для обращения к методам этого класса CLXDisplay API предоставляет несколько функций. В листинге 2.1 представлен вариант для Delphi Language.

Листинг 2.1. Функции CLXDisplay API для работы с классом QPushButton

```
// Конструкторы QPushButton
function QPushButton_create(parent: QWidgetH; name: PAnsiChar):
QPushButtonH; overload; cdecl;
function QPushButton_create(text: PWideString; parent: QWidgetH;
name: PAnsiChar): QPushButtonH; overload; cdecl;
function QPushButton_create(icon: QiconSetH; text: PWideString;
parent: QWidgetH; name: PAnsiChar): QPushButtonH; overload; cdecl;
// Деструктор QPushButton
procedure QPushButton_destroy(handle: QPushButtonH); cdecl;
// Методы QPushButton
procedure QPushButton_setFlat(handle: QPushButtonH; p1: Boolean);
cdecl;
function QPushButton_isDefault(handle: QPushButtonH): Boolean;
cdecl;
procedure QPushButton_setDefault(handle: QPushButtonH; def: Boolean);
cdecl;
...
```

Обратите внимание на директиву `overload`. В качестве префикса к имени функции используется имя класса, в данном случае `QPushButton`. Далее следует `_create`, означающее, что функция создает экземпляр класса и вызывает его конструктор. Все три функции возвращают значение типа `QPushButtonH`, которое является ссылкой на созданный экземпляр объекта. Это значение следует передавать другим функциям для работы с экземпляром `QPushButton`. Имя типа ссылки на экземпляр класса составляется из имени класса с добавлением `H`.

В листинге 2.2 представлена общая схема использования функций CLXDisplay API в приложении Delphi Language.

Листинг 2.2. Пример использования функций CLXDisplay API

```
uses
  Qt;
...
var
  PushButton : QPushButtonH
...
// Создаем экземпляр класса
PushButton := QPushButton_create(parent, nil);
...
// Вызываем метод класса для данного экземпляра
QPushButton_setFlat(PushButton, True);
...
// Уничтожаем экземпляр класса
QPushButton_destroy(PushButton);
...
```

Qt library и Kylix C++

Хотя в основных чертах взаимодействие с CLXDisplay API в Kylix C++ происходит также, как и в случае с Delphi Language, в реализации интерфейса CLXDisplay API для C++ существуют некоторые отличия. В Delphi Language ссылки на классы Qt представляют собой "пустые" классы Delphi. Иерархия этих классов-ссылок соответствует иерархии классов Qt library. Поэтому при работе в Delphi Language выполняется неявное преобразование типов *классов-потомков* к типам *классов-предков*. В файле Qt.hpp такие классы в стиле Delphi тоже определены, однако функции CLXDisplay API используют для ссылок на объекты другие типы. Это связано с тем, что в отличие от Delphi Language, C++ непосредственно импортирует C-функции CLXDisaplay API, и использовать в заголовках этих функций ссылки на классы Delphi, разумеется, *нельзя*.

Приведем простой пример. Функция `QLabel_create` создает экземпляр Qt-класса `QLabel`. В Kylix C++ эта функция возвращает значение типа `QLabel__`, а не `QLabelH`, как в Delphi Language. Тем не менее тип `QlabelH` определен в файле Qt.hpp и тип `QLabel__` приводим к типу `QLabelH`. Таким образом, в Kylix C++ возможны два подхода, которые демонстрируют листинги 2.3 и 2.4 соответственно.

Листинг 2.3. Использование типа `QLabelH`

```
#include <Qt.hpp>
...
QLabelH *Label = (QLabelH *) Qlabel_create(NULL, NULL, 0);
WideString S = "Мой текст";
QLabel_setText((QLabel__ *) Label, &S);
...
```

Листинг 2.4. Использование типа `QLabel__`

```
#include <Qt.hpp>
...
QLabel__ *Label = Qlabel_create(NULL, NULL, 0);
WideString S = "Мой текст";
QLabel_setText(Label, &S);
...
```

Первый вариант требует постоянного преобразования типов, зато используемые объекты соответствуют типам, принятым в среде Delphi Language.

Вторая особенность реализации CLXDisplay API связана с перегруженными методами. Парадоксально, но Kylix C++ не может использовать перегрузку функций CLXDisplay API, как это делается в Delphi Language. Объясняется это тем, что в C++ CLXDisplay API импортирует функции в формате C. В результате в файле Qt.hpp три конструктора класса `QPushButton` имеют имена: `QPushButton_create`, `QPushButton_create2` и `QPushButton_create3`.

Архитектура Qt library и CLXDisplay API

Как уже отмечалось, библиотека Qt library образует иерархию классов. Предком всех визуальных элементов Qt library является класс `QWidget`, в котором реализованы базовые методы и свойства графических элементов управления.

Не все классы Qt являются потомками `QWidget`. Многие классы непосредственно происходят от классов `QObject` и `QPaintDevice` (сам класс `QWidget` является потомком этих классов). Ссылки на классы в CLXDisplay API определены таким образом, что ссылку на класс-потомок всегда можно использовать как ссылку на класс-предок, но не наоборот. Таким образом, решается проблема с наследуемыми методами. Если нам, например, нужно вызвать для экземпляра класса `QPushButton` метод, унаследованный этим классом от `QWidget`, необходимо использовать функцию, инкапсулирующую

соответствующий метод класса `QWidget`. При этом ссылку на класс `QPushButton` — `QPushButtonH` или `QPushButton__` следует преобразовать в тип `QWidgetH` или `QWidget__` соответственно. Как видим, для того чтобы найти функцию CLXDisplay API, импортирующую нужный метод, необходимо знать, в каком из предков данного класса этот метод был впервые объявлен. Информацию об этом можно получить в справочной системе по Qt library, которая поставляется вместе с дистрибутивом ОС Linux.

Мы рассмотрели методы создания экземпляров Qt-классов при помощи функций CLXDisplay API. А как обращаться к уже существующим Qt-объектам, т. е. к тем, которые лежат в основе объектов VisualCLX, используемых в приложении? Для этого в каждом компоненте VisualCLX, основанном на Qt-объекте, реализовано свойство `Handle`, значение которого представляет собой не что иное, как ссылку на соответствующий объект Qt. Это значение и следует использовать при вызове функций CLXDisplay API для компонентов VisualCLX.

Основным механизмом взаимодействия между приложениями, основанными на Qt library, и операционной системой являются *события*. Каждому событию в Qt соответствует свой класс Qt library. Обычно все объекты-события Qt сначала передаются экземпляру класса `QApplication` (основа приложения Qt), который либо сам обрабатывает событие, либо передает его тому объекту, который должен его обрабатывать. Объект, получивший событие, определяет его тип и на основе этой информации вызывает обработчик, являющийся методом Qt-объекта. Обработчику события передается ссылка на объект-событие, из которого обработчик может получить необходимую информацию о событии. Для того чтобы определить собственный обработчик события для какого-либо Qt-класса в Qt library, следует создать производный класс этого Qt-класса и заместить (*override*) метод-обработчик соответствующего события. Однако Kylix не позволяет создавать производные Qt-классы и замещать их методы, поэтому в CLXDisplay API реализован специальный механизм обработки событий.

Важным механизмом взаимодействия между объектами Qt library в рамках одного приложения являются *сигналы* и *слоты*. Концепция сигналов Qt очень похожа на концепцию событий, но, в отличие от событий, сигналы являются не отдельными классами Qt, а элементами Qt-объектов. Как правило, объект Qt генерирует (в терминологии Qt: *имитирует*) сигнал в ответ на событие, однако сигналы могут посылаться и из обычного кода программы. Для приема и обработки сигналов используют слоты. Также как и сигналы, слоты являются частью объектов Qt. Каждому слоту в данном объекте сопоставлен какой-либо метод этого объекта. Для того чтобы объект реагировал на некоторый сигнал, необходимо связать этот сигнал с одним из слотов объекта. В этом случае после генерации сигнала будет вызван метод, соответствующий

данному слоту. Если сигнал несет какие-либо данные о событии, эти данные могут быть переданы методу-обработчику через его параметры.

Связывание сигналов и слотов похоже на назначение обработчиков событий объектов Delphi Language, однако между реализацией взаимодействия объектов в Delphi Language и Qt есть существенные различия. Во-первых, многие объекты библиотеки Qt library уже имеют слоты для обработки определенных сигналов, и для связывания их друг с другом не требуется перекрывать их методы в объектах-потомках. Во-вторых, механизм взаимодействия сигналов и слотов позволяет связывать сигналы и слоты разных типов, не заботясь о соответствии списков параметров. Если список параметров сигнала не соответствует списку параметров обработчика, при вызове обработчик получает значения параметров, установленные по умолчанию. Третье отличие заключается в возможности связывать один сигнал с несколькими слотами и один слот с несколькими сигналами. Это означает, что событию может быть сопоставлено несколько обработчиков, являющихся методами разных объектов, и в ответ на событие будут вызываться все назначенные ему обработчики.

Следует отметить одно ограничение, налагаемое CLXDisplay API на использование сигналов и слотов. В Qt library для добавления в класс нового сигнала или слота создается класс-потомок базового класса. В Kylix мы не можем создавать классы-потомки классов Qt, а значит, мы можем использовать только те сигналы и слоты, которые уже определены в базовых классах. (Впрочем, набора этих сигналов и слотов вполне достаточно для решения большинства задач.)

Если наше приложение использует исключительно объекты VisualCLX, нам не придется сталкиваться с событиями Qt, сигналами и слотами, так как объектная модель VisualCLX скрывает все эти механизмы. Однако, если мы хотим работать напрямую с классами Qt library при помощи CLXDisplay API, нам не обойтись без использования специфических средств Qt library. В следующих разделах мы рассмотрим использование этих механизмов на практике. Кроме того, следует отметить, что в VisualCLX реализованы не все возможности Qt library и даже не все возможности, предоставляемые CLXDisplay API. Например, для некоторых событий, связанных с визуальными элементами Qt library, не существует средств обработки на уровне VisualCLX. Если мы хотим, чтобы графические элементы нашей программы обрабатывали эти события, нам придется организовать их обработку на уровне CLXDisplay API.

Перед тем, как перейти к более подробному описанию методов работы с CLXDisplay API, следует отметить, что не только приложения, использующие исключительно компоненты VisualCLX, но и программы, прибегающие к функциям CLXDisplay API, как правило, являются переносимыми между средами разработки для ОС Windows и Linux.

Обработка событий Qt в Kylix-приложении

CLXDisplay API предоставляет Kylix-программистам два метода обработки событий Qt library: использование обработчика `OnEvent` класса `TApplication` и использование *перехватчиков событий* (event hooks).

Обработчик *OnEvent*

Если свойству `OnEvent` класса `TApplication` назначена процедура-обработчик, эта процедура вызывается для каждого события каждого объекта Qt, являющегося дочерним объектом формы приложения, а также для других событий приложения. Обработчик `OnEvent` вызывается до того, как событие будет обработано стандартным обработчиком.

В Delphi Language тип процедуры обработчика декларируется следующим образом:

```
procedure (Sender: QObjectH; Event: QEventH; var Handled: Boolean)
of object;
```

В параметре `Sender` передается ссылка на объект, который должен реагировать на событие. Параметр `Event` является указателем на объект события. Этот параметр имеет тип `QEventH`, соответствующий классу `QEvent` — базовому классу для всех классов-событий Qt library. Поскольку обработчик `OnEvent` вызывается для всех событий, в нем обязательно следует проверять, какой именно потомок класса `QEvent` был передан. Переменная `Handled` позволяет сообщить компоненту `VisualCLX` о том, следует ли вызывать обработчик события, определенный в компоненте.

Для C++ декларация процедуры-обработчика `OnEvent` выглядит так:

```
void __fastcall EHandler(Qt::QObjectH *Sender, Qt::QEventH *Event,
bool &Handled);
```

Также как и в случае с Delphi Language, процедура-обработчик должна быть методом объекта.

Рассмотрим практический пример применения обработчика событий. Каждый раз, когда окно приложения перемещается по экрану (например, перетаскивается мышью за заголовок окна), приложение получает событие `QMoveEvent`. У компонента `TForm` нет обработчика для этого события. Положение формы на экране всегда можно определить при помощи свойств `Top` и `Left`, но сам момент изменения этого положения формой никак не фиксируется. Самый простой способ научить приложение реагировать на событие `QMoveEvent` — назначить обработчик событию `OnEvent` класса `TApplication` и выполнить необходимые действия в этом обработчике.

В нашем примере при перемещении главного окна приложение, написанное на C++, будет выводить в своем заголовке новые координаты

своего верхнего левого угла. Заглавный файл приложения приводится в листинге 2.5.

Листинг 2.5. Заглавный файл главного модуля приложения MoveMe

```
//---------------------------------------------------------------------------

#ifndef MainH
#define MainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <Qt.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
 private:     // User declarations
 public:      // User declarations
   __fastcall TForm1(TComponent* Owner);
   void __fastcall EHandler(Qt::QObjectH *Sender, Qt::QEventH *Event,
   bool &Handled);
};
//---------------------------------------------------------------------------
#endif
```

Реализация класса приводится в листинге 2.6.

Листинг 2.6. Методы класса `TForm1` приложения MoveMe

```
//---------------------------------------------------------------------------
----
#include <clx.h>
#pragma hdrstop

#include "Main.h"
//---------------------------------------------------------------------------
```

```
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
  Application->OnEvent = &EHandler;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
Application->OnEvent = NULL;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::EHandler(Qt::QObjectH *Sender, Qt::QEventH
*Event, bool &Handled)
{
  if (Sender == this->Handle)
  if (QEvent_type((QEvent__*) Event) == QEventType_Move)
  {
    QMoveEvent__ *ME = (QMoveEvent__*) Event;
    TPoint *p = (TPoint*) QMoveEvent_pos(ME);
    Caption = Format("MoveMe window position: x=%d; y=%d",
    ARRAYOFCONST((p->x, p->y)));
  }
}
```

Для того чтобы определить, какому именно объекту соответствует событие, мы сравниваем значение `Sender` с полем `Handle` экземпляров объектов приложения. Далее нам следует определить, какое именно событие поступило. В предыдущих версиях Kylix для этого следовало использовать функции `QEvent__isQ*Event`, определенные в модуле `Qt.pas`. Аргументом этих функций является ссылка на объект `QEvent`. Функция возвращает `True`, если ей передан указатель на объект события соответствующего ей типа, и `False` — в противном случае.

В Kylix 3 появилась новая функция — `QEvent__type`, позволяющая определить тип события по переданной ссылке. Эта функция возвращает значение, соответствующее одной из констант `QeventType_*`, идентифицирующих событие.

После того как мы определили тип события, мы можем преобразовать значение параметра `Event` к соответствующему типу и при помощи функций CLXDisplay API получить доступ к методам объекта-события.

В нашем обработчике мы не изменяем значение переменной `Handled`, так как в случае события `QMoveEvent` эта переменная все равно игнорируется (дело в том, что событие `QMoveEvent` передается приложению уже *после* того, как форма была перемещена, и запрещать обработку события не имеет смысла). По умолчанию переменная `Handled` имеет значение `False`, и если в обработчике `OnEvent` это значение не будет изменено, событие будет обработано стандартными средствами. Для того чтобы запретить дальнейшую обработку события стандартными средствами, нужно присвоить этой переменной значение `True`.

Замечание

Изменять значение переменной `Handled` следует только после того, как мы определили тип события. Мы же не хотим запретить обработку всех событий приложения?

Хочется обратить внимание на присвоение значения `NULL` свойству `OnEvent` в методе `FormClose`. Так как время жизни объекта `TApplication` превышает время жизни объекта `TForm`, программа может пытаться вызвать обработчик `OnEvent` после уничтожения формы. Если после закрытия формы данное поле не будет помечено как *свободное* (`unassigned`), при выходе из программы возникнет ошибка сегментации.

Перехватчики событий

У обработчика `OnEvent` есть один серьезный недостаток: его использование замедляет работу приложения. Ведь будучи назначен, этот обработчик вызывается для каждого события, независимо от того, обрабатывается оно обработчиком или нет. Кроме того, если вы хотите обрабатывать несколько разных событий, код обработчика `OnEvent` может стать слишком громоздким. Зачастую при обработке событий Qt более предпочтительным является метод, использующий перехватчики событий.

Перехватчики событий подобны обработчику события `OnEvent`, с той разницей, что перехватчики назначаются для отдельных объектов Qt, причем каждому объекту может быть назначено несколько перехватчиков для обработки разных типов событий.

Перехватчики событий представляют собой специальные классы. Экземпляры классов-перехватчиков связывают объекты Qt, события которых следует обрабатывать, с обработчиками событий. Обработчик событий может быть процедурой или функцией, являющейся методом объекта Kylix и исполь-

зующей формат вызова `cdecl`. Каждому типу событий соответствует свой тип процедуры-обработчика. Обработчику передаются две ссылки:

1. На объект-событие.
2. На объект-источник события.

Если обработчик данного типа событий является функцией, он возвращает значение типа `Boolean`, указывающее на то, должен ли вызываться стандартный обработчик данного события или нет. Для того чтобы назначить перехватчик некоторого события какому-либо экземпляру класса Qt, необходимо создать экземпляр класса-перехватчика этого события для данного объекта Qt и связать его с методом-обработчиком.

В предыдущем разделе мы обрабатывали событие, на которое стандартные компоненты Kylix не реагируют. В этом разделе мы напишем приложение, позволяющее динамически отслеживать состояние буфера обмена Qt. Для этого мы, естественно, воспользуемся перехватчиками событий.

Состоянием буфера обмена Qt управляет класс `QClipboard`. Объект `QClipboard` — системный. Приложения Qt не создают и не уничтожают этот объект. Вместо этого каждому объекту `QApplication` передается ссылка на объект `QClipboard`. Получить ссылку на объект буфера обмена можно, например, при помощи функции `QApplication_clipboard`.

Каждый раз, когда какое-либо приложение изменяет содержимое буфера обмена Qt, объект `QClipboard` получает специальное событие. Для того чтобы отслеживать изменение состояния буфера, мы должны назначить обработчик этому событию.

В листинге 2.7 представлен исходный текст приложения ClipboardMonitor.

Листинг 2.7. Исходный текст приложения ClipboardMonitor

```
unit Main;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms,
  QDialogs, Qt, QStdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    FormatsMemo: TMemo;
```

```
    Label2: TLabel;
    DataMemo: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    CBHook : QClipboard_hookH;
    CB : QClipboardH;
    procedure ClipboardDataChanged; cdecl;
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.FormCreate(Sender: TObject);
var
  M : TMethod;
begin
  CB := QApplication_clipboard;
  CBHook := QClipboard_hook_create(CB);
  QClipboard_dataChanged_Event(M):=ClipboardDataChanged;
  QClipboard_hook_hook_dataChanged(CBHook, M);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  QClipboard_hook_destroy(CBHook);
end;

procedure TForm1.ClipboardDataChanged;
var
  QMS : QMimeSourceH;
```

```
  S : WideString;
  S1 : String;
  i : Integer;
begin
  QMS:=QClipboard_data(CB);
  FormatsMemo.Lines.Clear;
  (* Сканируем форматы данных в буфере обмена *)
  i:=0;
  S1:=QMimeSource_format(QMS, i);
  while S1<>'' do
  begin
    FormatsMemo.Lines.Add(S1);
    Inc(i);
    S1:=QMimeSource_format(QMS, i);
  end;
  DataMemo.Clear;
  (* Если буфер содержит данные в текстовом формате, отображаем их *)
  if QTextDrag_canDecode(QMS) then
  begin
    QTextDrag_decode(QMS, @S);
    DataMemo.Text := S;
  end;
end;
end.
```

Обработка событий буфера обмена выполняется в процедуре `ClipboardDataChanged`. В конструкторе формы мы сначала получаем ссылку на объект буфера обмена, а затем создаем объект-перехватчик для этого объекта. Объект-перехватчик создается при помощи функции `QClipboard_hook_create`. Эта функция — одна из многих функций создания объектов-перехватчиков, определенных в модуле Qt. Далее при помощи функции `QClipboard_hook_hook_dataChanged` мы связываем объект-перехватчик с процедурой-обработчиком.

Обратите внимание на преобразование типов. Типы процедур-обработчиков также определены в модуле `Qt`. В деструкторе формы мы уничтожаем экземпляр класса перехватчика.

В обработчике события мы сначала создаем список форматов данных, содержащихся в буфере в компоненте `FormatsMemo`, а затем, если буфер содержит данные в текстовом формате, выводим их в компонент `DataMemo`.

При помощи функции QClipboard_data мы получаем ссылку на объект QMimeSourceH, являющийся контейнером данных, содержащихся в буфере обмена. Этот объект позволяет также получить информацию о форматах данных, для чего используется функция QMimeSource_format. Эта функция возвращает строку с именем типа данных. Первый параметр функции — указатель на объект-контейнер, второй параметр — номер формата данных. Форматы нумеруются с нуля. Если значение номера формата превышает число форматов, возвращается пустая строка. Далее с помощью функции QTextDrag_canDecode мы проверяем, содержит ли объект-контейнер данные в текстовом формате и если содержит, извлекаем эти данные при помощи функции QTextDrag_Decode.

Результат работы программы ClipboardMonitor представлен на рис. 2.2.

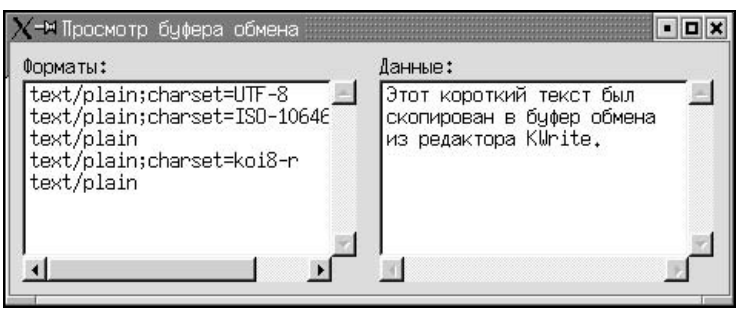


Рис. 2.2. Приложение ClipboardMonitor

У класса QApplication есть два метода — sendEvent и postEvent, позволяющие передавать объекты-события объектам приложения. Метод sendEvent вызывает обработчик события напрямую и возвращает управление только тогда, когда обработчик завершит обработку события. При этом значение, возвращенное sendEvent, соответствует значению, возвращенному обработчиком. Метод postEvent помещает событие в очередь и возвращает управление немедленно. Эти методы позволяют программе имитировать системные события для своих объектов. В CLXDisplay API методам sendEvent и postEvent соответствуют функции QApplication_sendEvent и QApplication_postEvent. Пример передачи текста в компонент TEdit с помощью события QKeyEvent рассмотрен в листинге 2.8.

Листинг 2.8. Передача события компоненту TEdit

```
WideString S = "Some text";
QKeyEvent__ * KE = (QKeyEvent__ *) QKeyEvent_create(
                    QEventType_KeyPress, 0, 0, 0, &S, true, S.Length());
QApplication_sendEvent((QObject__*) Edit1->Handle, (QEvent__ *) KE);
QKeyEvent_destroy(KE);
```

Сначала мы создаем объект события, указав в нем необходимые параметры, а затем передаем этот объект компоненту ввода. Поскольку объект-событие был передан при помощи функции `QApplication_sendEvent`, этот объект можно уничтожить. Если бы для передачи объекта события использовалась функция `QApplication_postEvent`, мы уже не имели бы контроля над этим объектом, и вызов для него функции `QKeyEvent_destroy` привел бы к ошибке сегментации.

Сигналы и слоты Qt library

Описание механизма сигналов и слотов см. в разд. "Архитектура Qtlibrary и CLXDisplay API" данной главы. Для связывания сигналов и слотов в CLXDisplay API служит функция `QObject_connect`. Этой функции передается четыре параметра. Первый параметр — указатель на объект-источник сигнала. Второй параметр — строка типа `PChar`. В этой строке передается имя сигнала, соответствующее синтаксису языка C++. Синтаксис конкретных сигналов и слотов, как и их описание, приводится в документации по Qt library. Имя сигнала должно предваряться символом "2". Третий параметр функции `QObject_connect` — указатель на метод-приемник. Четвертый параметр — строка типа `PChar`, содержащая имя слота, соответствующее синтаксису C++ и предваряемое символом "1". Если связывание сигналов и слотов прошло успешно, функция `QObject_connect` возвращает значение `True`, в противном случае — `False`.

Рассмотрим два простых примера использования сигналов и слотов. Пусть в нашем приложении есть три компонента ввода `TEdit` и кнопка `TButton`. Следующий код (листинг 2.9), связывает кнопку и строки ввода таким образом, что при нажатии на кнопку все компоненты "строка ввода" очищаются, а при вводе текста в компоненте `Edit1` этот текст дублируется в двух других компонентах.

Листинг 2.9. Пример использования сигналов и слотов

```
if (!QObject_connect((QObject__ *) ClearAllButton->Handle, "2pressed()",
     (QObject__ *) Edit1->Handle, "1clear()") ||
     !QObject_connect((QObject__ *) ClearAllButton->Handle, "2pressed()",
     (QObject__ *) Edit2->Handle, "1clear()") ||
     !QObject_connect((QObject__ *) ClearAllButton->Handle,
     "2pressed()", (QObject__ *) Edit3->Handle, "1clear()"))
ShowMessage("Ошибка при связывании с ClearAllButton");


if (!QObject_connect((QObject__ *) Edit1->Handle,
```

```
    "2textChanged(const  QString &)", (QObject__ *) Edit2->Handle,
    "1setText(const QString &)") ||
    !QObject_connect((QObject__ *) Edit1->Handle,
    "2textChanged(const QString &)", (QObject__ *) Edit3->Handle,
    "1setText(const QString &)"))
ShowMessage("Ошибка при связывании с Edit1");
```

Здесь `ClearAllButton` — объект класса `TButton`, а `Edit1`, `Edit2` и `Edit3` — элементы ввода.

Для разрыва связи между сигналом и слотом служит функция `QObject_disconnect`.

Примеры использования CLXDisplay API

Рассмотрим еще несколько важных примеров расширения возможностей Kylix-приложений при помощи CLXDisplay API. Первый пример — приложение, позволяющее обмениваться данными с другими программами методом Drag and Drop ("Перетащить и бросить"). Средства Drag and Drop, реализованные в VisualCLX позволяют обмениваться данными только между разными компонентами одного приложения, и для реализации "полномасштабного" Drag and Drop нам придется прибегнуть к CLXDisplay API.

В приложении-примере Drag and Drop Demo в роли активного элемента управления Drag and Drop будет выступать компонент `DragDropLabel` класса `TLabel`. Мы сможем перетаскивать текст этого компонента в другие приложения или перетаскивать текст из других приложений на этот компонент. Для того чтобы компонент `DragDropLabel` корректно обрабатывал события `Qt Drag and Drop`, мы назначим ему перехватчик событий. В листинге 2.10 приводится исходный текст приложения Drag and Drop Demo.

Листинг 2.10. Приложение Drag and Drop Demo

```
unit Main;

interface

uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls, Qt;

type
  EventFunc = function(Handle : QObjectH; e : QEventH) : Boolean
```

```
    of object; cdecl;

  TForm1 = class(TForm)
    DragDropLabel: TLabel;
    procedure DragDropLabelMouseMove(Sender: TObject; Shift: TShiftState;
      X, Y: Integer);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
    DropHook : QEvent_hookH;
    function EventHandler(Handle : QObjectH; e : QEventH) : Boolean;
      cdecl;
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

function TForm1.EventHandler;
var
  QMS : QMimeSourceH;
  S : WideString;
begin
  Result:=False;
  case QEvent_type(e) of
    QEventType_DragEnter:
    begin
      if QDropEvent_source(QDropEventH(e))=DragDropLabel.Handle
      then Exit;
      QMS:=QDropEvent_to_QMimeSource(QDropEventH(e));
      QDropEvent_acceptAction(QDropEventH(e), QTextDrag_canDecode(QMS));
    end;
```

```
    QEventType_DragMove:
    begin
      if QDropEvent_source(QDropEventH(e))=DragDropLabel.Handle
      then Exit;
      QMS:=QDropEvent_to_QMimeSource(QDropEventH(e));
      QDropEvent_acceptAction(QDropEventH(e), QTextDrag_canDecode(QMS))
    end;
    QEventType_Drop:
    begin
      QMS:=QDropEvent_to_QMimeSource(QDropEventH(e));
      if QTextDrag_canDecode(QMS) then
      QTextDrag_decode(QMS, @S);
      DragDropLabel.Caption:=S;
    end;
  end;
end;

procedure TForm1.DragDropLabelMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var
  TextDragObject : QTextDragH;
  S : WideString;
  name : PChar;
begin
  if ssLeft in Shift then
  begin
    S:=DragDropLabel.Caption;
    name:='textdrag';
    TextDragObject:=QTextDrag_create(@S, DragDropLabel.Handle, name);
    if QDragObject_drag(TextDragObject) then DragDropLabel.Caption:='';
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  M : TMethod;
begin
  DropHook:=QEvent_hook_create(DragDropLabel.Handle);
```

```
  EventFunc(M):=EventHandler;
  Qt_hook_hook_events(DropHook, M);
end;


procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  QEvent_hook_destroy(DropHook);
end;
end.
```

Обработчиком событий Drag and Drop является функция `EventHandler`. Назначение объекту `DragDropLabel` перехватчика и обработчика выполняется аналогично рассмотренному в листинге 2.7, с той лишь разницей, что в этом случае для преобразования типов нам приходится декларировать специальный тип `EventFunc`.

Рассмотрим подробнее текст функции-обработчика. В нашем обработчике мы обрабатываем события трех типов:

1. `QEventType_DragEnter` — курсор в режиме Drag and Drop входит в область компонента `DragDropLabel`.
2. `QEventType_DragMove` — курсор в режиме Drag and Drop перемещается в области компонента `DragDropLabel`.
3. `QEventType_Drop` — курсор в режиме Drag and Drop отпускает объект в области компонента `DragDropLabel`.

Функция `QDropEvent_source` вызывает метод `source` объекта класса `QDropEvent` — базового класса событий Drag and Drop. Обратим внимание на преобразование типа аргумента функции. Такое преобразование корректно, поскольку объекты `QDragEnterEvent` и `QDragMoveEvent` являются потомками объекта `QDropEvent`. `QDropEvent_source` позволяет определить источник объекта Drag and Drop. Поскольку в создаваемом приложении компонент `DragDropLabel` является и приемником и источником объектов Drag and Drop, мы должны проверять, какой Qt объект создал данный объект Drag and Drop и не выполнять дальнейшую обработку события, если источником объекта является компонент `DragDropLabel`.

Функция `QDropEvent_to_QMimeSource` преобразует объект класса `QDropEvent` в объект класса `QMimeSource`, являющийся контейнером передаваемых данных. Функция `QDropEvent_acceptAction` позволяет подтвердить операцию Drag and Drop, или отказаться от нее. Первый аргумент функции — ссылка на объект `QDropEvent`, второй аргумент — значение `True` или `False`. Для определения того, следует ли обрабатывать событие, мы используем функцию `QTextDrag_canDecode`. Эта функция возвращает `True`, если переданный ей объект `QMimeSource` содержит данные в текстовом формате, и `False` в

противном случае. Функция `QTextDrag_decode` извлекает текстовые данные из объекта `QMimeSource`. Обратите внимание, что обработчик не должен пытаться уничтожить объекты `QMimeSource`

Для того чтобы Qt-приложение могло стать источником объектов Drag and Drop, необходимо создать объект-контейнер данных (в нашем случае это объект класса `QTextDrag`) и вызвать один из методов этого объекта: `drag` или `dragCopy`. Объект для перетаскивания лучше всего создавать в момент начала движения мыши при нажатой левой кнопке. В нашем приложении эту функцию выполняет обработчик события `OnMouseMove` объекта `DragDropLabel`. После того, как объект-контейнер создан, мы вызываем функцию `QDragObject_drag`, которая и контролирует процесс "перетаскивания". Функция возвращает управление только после того, как перетаскиваемый объект будет отпущен, причем, если операция Drag and Drop завершилась успешно, функция возвращает значение `True`, а если нет — `False`.

Еще один пример — использование в Kylix-приложении класса `QRegExp`. Класс позволяет сравнить строку с образцом, содержащим регулярные выражения. Подобный компонент отсутствует в наборе компонентов CLX, по этому использование класса `QRegExp` может оказаться очень полезным для приложений, использующих регулярные выражения. Напишем демонстрационную функцию `MatchPattern` (листинг 2.11).

Листинг 2.11. Функция, демонстрирующая использование класса `QRegExp`

```
bool MatchPattern(WideString * Pattern, WideString * Text, bool
CaseSensitive)
{
  int dummy, res;
  QRegExp__ * RegExp = QRegExp_create1(Pattern, CaseSensitive, true);
  res = QRegExp_match(RegExp, Text, 0, &dummy, true);
  QRegExp_destroy(RegExp);
  return res !=0;
}
```

Параметр `Pattern` ссылается на образец, а параметр `Text` — строку для сравнения. Параметр `CaseSensitive` указывает, должно ли сравнение проводиться с учетом регистра символов или нет.

Например, вызов:

```
WideString Pattern = "* = [0-9].[0-9]";
WideString Text = "a = 2.7";
result = MatchPattern(Pattern, Text, false)
```

вернет значение `True`, а вызов:

```
WideString Pattern = "Hello";
WideString Text = "hello";
result = MatchPattern(Pattern, Text, false)
```

вернет значение `False`.

Запуск дочерних процессов из приложений с графическим интерфейсом

Запуск дочернего графического процесса из родительского консольного проблем не вызывает, а вот запуск дочернего графического процесса из графического приложения связан с некоторыми дополнительными сложностями. Запуск любого дочернего процесса в ОС Linux неизбежно включает в себя "клонирование" запускающего процесса, а два процесса не могут разделять между собой одни и те же ресурсы, предоставляемые графическим менеджером. Методы создания дочерних процессов, приводимые в литературе, не отличаются надежностью и устойчивостью.

Суть метода, который предлагается здесь, заключается в том, чтобы "раздвоить" процесс с помощью `fork` до того, как будут вызваны какие-либо графические функции. Затем родительский процесс устанавливает связь с графическим сервером, а дочерний — нет. Таким образом, мы получаем дочерний консольный процесс, запущенный из графического процесса. Как уже отмечалось выше, запустить графическое приложение из консольного процесса довольно просто, но мы хотим получить возможность запускать любое количество графических приложений в любой момент работы родительского графического процесса. Для этого нашу схему придется еще немного усложнить.

Между дочерним и родительским процессами устанавливается однонаправленный канал передачи данных, по которому родительский процесс передает дочернему команду на запуск нового процесса. Дочерний процесс постоянно находится в режиме ожидания команды родительского процесса. Как только команда поступает, дочерний процесс снова "раздваивается" и уже вторичный дочерний процесс выполняет переданную команду с помощью функции `execlp`.

Описанный метод демонстрируется в приложении `runcmd` (листинги 2.12, 2.13, 2.14).

Листинг 2.12. Файл runcmd.cpp

```
//---------------------------------------------------------------------------


#include <clx.h>
```

```
#pragma hdrstop
USEFORM("Main.cpp", Form1);
//---------------------------------------------------------------------
int pipedes[2];
int pid;

int main(void)
{
        pipe(pipedes); // Создаем канал для передачи команд
        pid = fork(); // Раздваиваем процесс
        if (pid == 0)
        {
           // дочерний процесс ожидает команду
          char buf[256];
          while (true)
          {
            int len = read(pipedes[0], buf, 256);
            buf[len] = 0;
            pid_t chpid = fork();
            if (chpid == 0)
            execlp(buf, " ", NULL); // Вторичный дочерний процесс
          }
        }
        // Родительский процесс (графический)
        try
        {
          Application->Initialize();
          Application->CreateForm(__classid(TForm1), &Form1);
          Application->Run();
        }
        catch (Exception &exception)
        {
          Application->ShowException(&exception);
        }
        catch(...)
        {
          try
          {
```

```
            throw Exception("");
          }
          catch(Exception &exception)
          {
            Application->ShowException(&exception);
          }
        }
        return 0;
}
//---------------------------------------------------------------------------
```

Листинг 2.13. Файл Main.h

```
//---------------------------------------------------------------------------

#ifndef MainH
#define MainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <QDialogs.hpp>
#include <QFileCtrls.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TButton *ExecButton;
        TButton *BrowseButton;
        TOpenDialog *OpenDialog1;
        TLabel *Label1;
        TFileEdit *CommandEdit;
        void __fastcall ExecButtonClick(TObject *Sender);
        void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
        void __fastcall BrowseButtonClick(TObject *Sender);
        void __fastcall CommandEditKeyPress(TObject *Sender, char &Key);
private:      // User declarations
public:       // User declarations
```

```
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Листинг 2.14. Файл Main.cpp

```
//---------------------------------------------------------------------------
#include <clx.h>
#include <unistd.h>
#pragma hdrstop
#include "Main.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
extern int pipedes[2];
extern int pid;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::ExecButtonClick(TObject *Sender)
{
  String S = (String) CommandEdit->Text;
  write(pipedes[1], &S[1], S.Length());
}
//---------------------------------------------------------------------------

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
  kill(pid, SIGKILL);
}
//---------------------------------------------------------------------------
```

```
void __fastcall TForm1::BrowseButtonClick(TObject *Sender)
{
  if (OpenDialog1->Execute())
  CommandEdit->Text = OpenDialog1->FileName;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::CommandEditKeyPress(TObject *Sender, char &Key)
{
  if (Key == '\r') ExecButtonClick(this);
}
//---------------------------------------------------------------------------
```

Передача команды запуска нового процесса выполняется в обработчике `ExecButtonClick`. Обращаем внимание на совместное использование переменных `pipedes` и `pid` двумя модулями. В деструкторе формы дочерний процесс уничтожается при помощи функции `kill`.

Кнопка **Выбрать** (`BrowseButton`) вызывает диалоговое окно **Открыть файл** (`FileOpen`), облегчающее выбор команды. Обратите внимание также, что в этом примере мы используем новый компонент Kylix — `TFileEdit`.

Для создания приложения runcmd следует создать в Kylix C++ IDE проект графического приложения, сохранить его под именем runcmd, и в файле runcmd.cpp добавить код содержащийся в листинге 2.9. Файл главного модуля должен называться Main.cpp.

Пример работы приложения runcmd приводится на рис. 2.3.

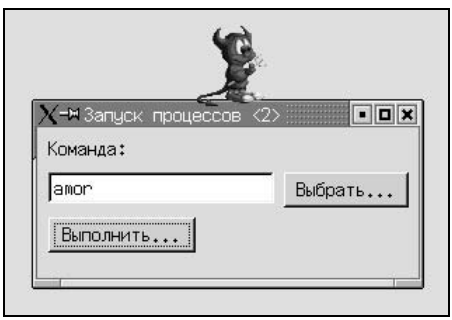


Рис. 2.3. Запуск команды `amor` из приложения runcmd

Для запуска нового процесса из приложения runcmd следует указать команду в строке ввода (компонент `TFileEdit`). При этом можно воспользоваться кнопкой **Выбрать** (`BrowseButton`). После этого следует нажать клавишу <Enter> или кнопку **Выполнить** (`ExecButton`).

Использование в Kylix-приложениях других графических интерфейсов

Возможности разработки графических приложений в Kylix не ограничиваются системой Qt library. Из-за несовместимости объектного кода классов GNU C++ и Kylix C++ в Kylix нельзя использовать непосредственно графические библиотеки ОС Linux, экспортирующие классы. Нельзя, например, использовать библиотеки пользовательского интерфейса KDE, такие как `libkdeui.so`. Так что вывести пиктограмму Kylix-приложения в tray-область KDE не получится. Однако в Kylix можно использовать все графические библиотеки ОС Linux, экспортирующие интерфейс в виде набора функций C. В этом разделе мы рассмотрим два примера использования графических интерфейсов в Kylix-приложении: непосредственное взаимодействие с интерфейсом X-Window и построение Kylix-приложения при помощи графического набора `gtk`.

Использование функций X-Window

В начале этой главы говорилось о том, что любое графическое приложение может обращаться непосредственно к функциям X-сервера. Существует две причины, по которым программисты обычно не делают этого. Во-первых, использование функций X-Window снижает кросс-платформенность создаваемого кода. Во-вторых, интерфейс программирования X-Window довольно сложен и не предоставляет непосредственно многих важных функций для построения пользовательского интерфейса приложения. Тем не менее иногда может понадобиться использовать отдельные функции X-Window в приложении или даже написать всю программу целиком с использованием этих функций.

Как уже отмечалось, практически каждому элементу управления VisualCLX соответствует X-окно. Для того чтобы вызвать функцию X-Window для некоторого окна, как правило, необходимо знать два параметра: указатель на соответствующий окну X-дисплей и идентификатор самого окна. CLXDisplay API позволяет получить и то и другое. Указатель на X-дисплей хранится в переменной `QtDisplay` из модуля `Qt`. Получить идентификатор X-окна элемента управления можно, воспользовавшись свойством `Handle` соответствующего объекта VisualCLX. Напомним, что свойство `Handle` содержит ссылку на "нижележащий" объект одного из классов Qt library. Если соответствующий класс Qt library является потомком класса QWidget, значит ему соответствует некоторое X-окно. Получить идентификатор этого окна можно с помощью функции CLXDisplay API `QWidget_winId`:

```
int WinId = QWidget_winId((QWidget__ *) Component->Handle);
```

В Delphi Language функции интерфейса X-Window объявлены в модуле `XLib`. Листинг 2.15 представляет программу "Зеленый квадрат", написанную с применением исключительно функций X-Window.

Листинг 2.15. X-приложение "Зеленый квадрат"

```
program XDemo;

uses
  Xlib;

var
  dp : PDisplay;
  wnd : Window;
  report : XEvent;
  gr_gc : TGC;
  color : XColor;
  clrmap : Colormap;
  green : PChar = '#00FF00';

  function XSetStandardProperties(display: PDisplay; w: TWindow;
  const window_name, icon_name: PChar; icon_pixmap: TPixmap;
  argv: PPChar; argc: LongInt; hints: PXSizeHints): LongInt;
  cdecl; external Xlibmodulename;

begin
  dp := XOpenDisplay(nil);
  wnd := XCreateSimpleWindow(dp, XRootWindow(dp, 0), 1, 1, 180, 180, 0,
XBlackPixel (dp, 0), XBlackPixel(dp, 0));
  XMapWindow(dp, wnd);
  XSetStandardProperties(dp, wnd, 'Kylix X-Demo', nil, 0, nil, 0, nil);
  clrmap := XDefaultColormap(dp, 0);
  gr_gc := XCreateGC(dp, wnd, 0, 0);
  XSelectInput(dp, wnd, ExposureMask or KeyPressMask or ButtonPressMask);
  XParseColor(dp, clrmap, green, @color);
  XAllocColor(dp, clrmap, @color);
  XFlush(dp);
  XSetForeground(dp, gr_gc, color.pixel);
  while True do
  begin
    XNextEvent(dp, @report);
    case report.xtype of
      Expose:
```

```
      begin
        XClearWindow(dp, wnd);
        XFillRectangle(dp, wnd, gr_gc, 10, 10,
        report.xexpose.width-20, report.xexpose.height-20);
        XFlush(dp);
      end;
      KeyPress:
      begin
        if XLookupKeysym(@report.xkey, 0) = 113 then
        Break;  // Вы можете нажать [q] для выхода из программы
      end;
    end;
  end;
end.
```

Описание интерфейса программирования X-Window выходит за рамки этой книги. Отметим здесь лишь один нюанс. Весьма полезная функция `XSetStandardProperties` почему-то не импортируется модулем `XLib`, и нам приходится импортировать эту функцию самим.

Результат выполнения программы "Зеленый квадрат" представлен на рис. 2.4.

Рис 2.4. Приложение "Зеленый квадрат"

Использование набора gtk

Функции gtk экспортируются в формате C и могут использоваться при программировании Kylix-приложений. В листинге 2.16 представлено Kylix-приложение, использующее интерфейс gtk 1.2. Исходный текст приложения взят из документации по программированию gtk, при этом в него внесены дополнения, необходимые для успешной компиляции приложения в Kylix C++ IDE.

Листинг 2.16. Приложение Kylix, использующее интерфейс gtk

```
#ifdef __BORLANDC__
#define G_INLINE_FUNC __inline
#endif

#include <gtk/gtk.h>
#include <gdk/gdk.h>

#pragma link "/usr/lib/libgdk.so"
#pragma link "/usr/lib/libgtk.so"
#pragma link "/usr/lib/libm.so"
#pragma link "/usr/lib/libdl.so"
#pragma link "/usr/lib/libglib.so"
#pragma link "/usr/lib/libgmodule.so"
#pragma link "/usr/X11R6/lib/libXi.so"
#pragma link "/usr/X11R6/lib/libXext.so"
#pragma link "/usr/X11R6/lib/libX11.so"

void hello(GtkWidget *widget,
           gpointer   data)
{
    g_print ("Hello World\n");
}

gint delete_event(GtkWidget *widget,
                  GdkEvent  *event,
                  gpointer   data)
{
    g_print ("delete event occurred\n");
    return(TRUE);
}

void destroy(GtkWidget *widget,
             gpointer   data)
```

```
{
    gtk_main_quit();
}
int main(int   argc,
         char *argv[])
{
    GtkWidget *window;
    GtkWidget *button;

    gdk_init(&argc, &argv);
    gtk_init(&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                        GTK_SIGNAL_FUNC (delete_event), NULL);
    gtk_signal_connect (GTK_OBJECT (window), "destroy",
                        GTK_SIGNAL_FUNC (destroy), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    button = gtk_button_new_with_label ("Hello World");
    gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (hello), NULL);
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC (gtk_widget_destroy),
                               GTK_OBJECT (window));
    gtk_container_add (GTK_CONTAINER (window), button);
    gtk_widget_show (button);
    gtk_widget_show (window);
    gtk_main ();
    return(0);
}
```

Для того чтобы скомпилировать это приложение в Kylix C++ IDE, необходимо выполнить следующие действия:

1. Сохранить текст листинга 2.16 в файле gtkdemo.c.
2. Создать заготовку консольного приложения Kylix. В окне **Console Wizard** выбрать в качестве языка проекта C и установить флажок **Specify Project Source**, затем выбрать в качестве источника файл gtkdemo.c.

3. Открыть окно **Project Options** и на вкладке **Directories/Conditionals** в строке **Include Path** добавить каталоги:
   - /usr/include/gtk-1.2;
   - /usr/include/glib-1.2;
   - /usr/lib/glib/include;
   - /usr/X11R6/include.

Если все сделано правильно, после компиляции проекта должна получиться программа, показанная на рис. 2.5.

Рис. 2.5. Демонстрационная gtk-программа

Часть II

Интернет-программирование

Глава 3

Принципы разработки интернет-приложений в Kylix 3

В предыдущих главах мы рассмотрели основы программирования в ОС Linux. С этой главы мы начинаем рассмотрение программирования важного типа приложений — приложений для Интернета. Эта глава, первая в данной части, посвящена обзору типов интернет-приложений, для разработки которых Kylix 3 предоставляет специальные средства, а также сторонние программные продукты, с которыми придется иметь дело разработчику интернет-приложений.

Типы интернет-приложений в Kylix 3

Интернет-приложения, создаваемые в Kylix 3, можно классифицировать как по типу исполняемого модуля, так и по используемым в них технологиям.

По типу исполняемого модуля приложения Kylix можно разделить на следующие категории:

- самостоятельные приложения;
- CGI-приложения;
- разделяемые модули для сервера Apache;
- приложения для отладчика Web App Debugger.

Последняя категория приложений отличается тем, что носит в некотором смысле "промежуточный" характер. Действительно, едва ли нашей целью является создание приложений для встроенного отладчика. Отладив приложение в среде Web App Debugger, мы, скорее всего, захотим преобразовать его к какому-либо другому типу.

Используемые при разработке технологии позволяют условно разделить создаваемые в Kylix приложения на следующие классы:

- приложения на основе Indy;
- приложения на основе WebBroker;
- приложения WebServices;
- приложения WebSnap.

Условный характер этой классификации связан с тем, что многие из перечисленных выше технологий можно использовать совместно в рамках одного приложения. Следует отметить также, что в принципе интернет-приложения Kylix могут быть написаны и без использования указанных технологий.

Рассмотрим подробнее перечисленные выше классы приложений. Независимые интернет-приложения с точки зрения системы являются обычными программными файлами. Эти приложения используют сетевой интерфейс ОС Linux. Обычно при программировании таких приложений в Kylix применяются компоненты Internet Direct (Indy).

CGI-приложения являются программами, предназначенными для взаимодействия с Web-сервером. Для обмена данными эти приложения используют не сетевой интерфейс, а функции сервера. В процессе разработки этого типа приложений в Kylix, как правило, применяются технологии WebBroker, WebServices и WebSanp.

По принципам своей работы разделяемые модули для сервера Apache (Apache DSO) очень похожи на CGI-приложения. Основное отличие заключается в том, что модули Apache представляют собой разделяемые библиотеки ОС Linux. Как явствует из их названия, приложения этого типа предназначены специально для Web-сервера Apache и позволяют использовать специфическую функциональность этого сервера. Как и CGI-приложения, модули Apache не обращаются непосредственно к функциям сетевого интерфейса и широко используют технологии WebBroker, WebServices и WebSanp.

Как следует из вышесказанного, CGI-приложения, модули Apache и приложения встроенного Web-отладчика можно объединить в одну категорию — приложения для Web-сервера. Приложения для Web-сервера расширяют его функциональность. Благодаря этим приложениям Web-сервер можно наделить многими функциями, присущими другим приложениям ОС Linux. Независимо от того, какую из Web-технологий мы хотим взять за основу нашего серверного приложения, Kylix предоставляет нам возможность выбрать любой из трех типов приложений для сервера. Выбрать тип приложения можно в специальном диалоговом окне, которое открывается при создании заготовки проекта (рис. 3.1).

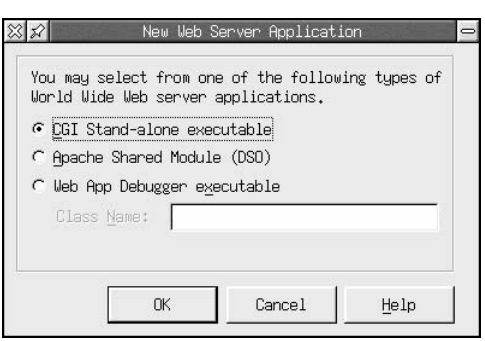


Рис 3.1. Окно **New Web Server Application**

Разработке приложений для Web-сервера посвящены главы 4, 6—9. Здесь же следует сказать несколько слов о самих Web-серверах.

Сервер Apache и другие

Apache — свободно распространяемый Web-сервер, разработка которого началась в 1995 году и продолжается по сей день. Web-сервер Apache портирован на множество платформ, включая ОС Linux, все современные версии ОС Unix, Mac OS, OS/2 и все 32-битные версии ОС Windows. Сегодня Apache — наиболее часто используемый Web-сервер (до 60% Web-серверов в Интернете используют Apache). Причинами популярности Apache у Web-администраторов стали не только открытость, надежность и простота Web-сервера, но и чрезвычайно низкая требовательность к аппаратным ресурсам. Минимальная конфигурация, необходимая для использования Apache — процессор 80386 и 2 Мбайт оперативной памяти. При этом Apache поддерживает практически все современные Web-технологии.

Неудивительно, что сервер Apache является фактическим стандартом в мире ОС Linux. Все дистрибутивы ОС Linux включают Apache. Если в процессе установки системы мы выберем пункт **Установка WWW-сервера**, можно не сомневаться, что в нашей системе будет установлен именно сервер Apache. Сервер Apache так тесно "дружит" с ОС Linux, что в этой системе программа носит название httpd, т. е. "демон протокола HTTP" (о программах-демонах речь пойдет в следующей главе).

Если же у нас нет сервера Apache или мы хотим использовать новейшую версию, мы можем загрузить либо исходные тексты с сайта **http://www.apache.org/,** либо найти подходящий двоичный дистрибутив по адресу **http://www.apache.org/dist/binaries/**.

Широкая распространенность Web-сервера Apache приводит к тому, что многие разработки для этого сервера тоже являются кросс-платформенными. В частности приложения, созданные для сервера Apache в Borland Kylix, очень легко перенести на платформу Windows с помощью Delphi и C++ Builder.

Подробное описание установки и настройки сервера Apache выходит за рамки этой книги (соответствующую информацию можно найти в подробном руководстве, поставляемом вместе с дистрибутивом сервера). Самый простой способ установить и настроить сервер Apache в ОС Linux — выбрать соответствующий пункт в процессе установки самой ОС.

Для того чтобы проверить, запущен ли сервер Apache в системе, достаточно набрать в окне Web-браузера адрес **http://localhost**. Если на открывшейся странице мы увидим пиктограмму с индейским пером, значит сервер работает. Получить информацию о текущих параметрах сервера можно в режиме root при помощи команд `apachectl status` и `apachectl statusextended`.

Каталоги, в которых хранятся файлы контента, предоставляемого Web-сервером удаленным клиентам, могут различаться в разных дистрибутивах ОС Linux. В некоторых дистрибутивах корневым каталогом для Web-контента может быть /usr/local/apache/htdocs/, в других — /var/www/. Некоторые системы могут быть настроены таким образом, что в домашнем каталоге каждого пользователя будет создан специальный подкаталог для Web-контента.

При использовании сервера Apache для передачи русскоязычного контента традиционно возникают проблемы, связанные с существованием нескольких русских HTML-кодировок. Для решения этих проблем Алексом Тутубалиным (**http://www.lexa.ru/lexa/**) был разработан русский сервер Apache, представляющий собой модификацию оригинального сервера. Вот как сам автор объяснил необходимость в разработке такого продукта: "На сегодняшний день в русскоязычном Internet-пространстве существует... по меньшей мере пять распространенных кодировок (кодовых таблиц) кириллицы, и "информационные ресурсы" должны доставляться потребителям в доступной для них форме, т.е. в той кодировке, которую поддерживает программное обеспечение пользователя. Как следствие, на русскоязычном WWW-сервере должна быть (или крайне желательна) поддержка нескольких кодировок кириллицы. ... Желательно было сделать такую поддержку максимально прозрачной для пользователя и гибкой в настройке для Web-мастера".

К основным особенностям русифицированного варианта сервера Apache следует отнести:

- поддержку согласования кодировок клиента и сервера как при отправке документов пользователю, так и при обработке пользовательского CGI-запроса (поддерживаются методы запроса `GET` и `POST`);
- выдачу корректных сведений в полях `Content-type:...;charset=...` в соответствии с указанным согласованием кодировок;
- генерацию при необходимости заголовка `Expires`: для proxy-серверов.
- генерацию корректных заголовков `Vary:` и `ETag`, в результате чего становится возможным корректное кэширование документов (если proxy-cache совместим с HTTP/1.1);
- автоматическое перенаправление клиента на URL в нужной кодировке.

Исходные тексты русского сервера Apache сопровождаются довольно подробной документацией по установке и настройке сервера, а также списком ответов на часто задаваемые вопросы.

Как бы ни был хорош сервер Apache, это не единственный сервер в мире ОС Linux. Среди других стоит отметить Sun^{tm} ONE Web Server (бывший Netscape IPlanet Web Server).

К достоинствам этого продукта следует отнести простоту установки и использования, мощную поддержку технологий Java и удобную возможность конфигурации сервера через Web-интерфейс. Следует отметить, что в отличие от сервера Apache, Sun^{tm} ONE Web Server — коммерческий продукт.

Среди некоммерческих альтернатив серверу Apache заслуживает внимания WN Server (**http://hopf.math.nwu.edu/**). Основными особенностями этого сервера являются:

- поддержка CGI 1.1;
- встроенная возможность поиска Web-страниц по заданным заголовкам и ключевым словам в заголовках (в форме поиска допускается использование регулярных выражений);
- встроенная возможность поиска Web-страниц по заданным ключевым словам в тексте страниц (в форме поиска допускается использование регулярных выражений);
- встроенная возможность поиска по заданным ключевым словам текстовых документов в определенных каталогах (в форме поиска допускается использование регулярных выражений);
- возможность подключения к серверу поисковых программ, выполняющих индексированный поиск;
- развитая возможность использования механизма включения документов на стороне сервера (Server-Side Includes, SSI). При этом возможно использование конструкций `if – else – endif`;
- возможность подключения сторонних программ-фильтров (например, утилиты ОС Linux `zcat`), преобразующих контент перед отправкой его клиенту.

Технологии и типы интернет-приложений

Наверное, у всякого программиста, знакомящегося с многообразием технологий разработки интернет-приложений в Borland Kylix, неизбежно возникает вопрос: какие средства выбрать?

Безусловно, выбор технологий разработки зависит от специфики конкретного проекта.

С помощью компонентов Indy Kylix позволяет достаточно просто создавать полноценные Web-сервера. Однако прежде чем браться за разработку собственного сервера, стоит подумать о его необходимости. Действительно, ОС Linux предоставляет в наше распоряжение прекрасный бесплатный сервер Apache. В пользу высокой надежности и безопасности сервера Apache говорит не только то, что этот сервер разработан чрезвычайно опытными программистами, но и его широкое распространение. Сотни тысяч людей используют сервер Apache по всему миру, а, значит, ошибки и "прорехи" в безопасности этого сервера выявляются и исправляются очень быстро. Возможность интеграции собственных приложений с сервером Apache позволяет нам наделить Web-сервер практически любой функциональностью. Та-

ким образом, в большинстве случаев разработка собственного сервера рискует стать попыткой "изобрести велосипед", причем скорее всего не самого высшего качества.

Вообще разработка собственных самостоятельных интернет-приложений оправдана либо в том случае, когда функциональность существующих приложений нас категорически не удовлетворяет, либо когда приложение должно стать частью некоторого интегрированного пакета. И то, и другое соображение в большей степени применимо к приложениям-клиентам, нежели к серверам, которые и так снабжены широкими средствами расширения функциональности и интеграции.

Что касается разработки приложений для сервера, то здесь прежде всего приходится выбирать между CGI-приложениями и разделяемыми модулями. У каждого из этих подходов есть свои преимущества.

Стандарт CGI поддерживается всеми профессиональными Web-серверами, а значит, CGI-приложение сможет работать совместно с любым Web-сервером на данной платформе. Выполняясь как независимые процессы, CGI-приложения, в случае возникновения в них ошибок, едва ли смогут нарушить нормальную работу всего Web-сервера. В то же время разделяемые модули часто обладают более высоким быстродействием по сравнению с CGI-приложениями, и позволяют использовать некоторые функции сервера, недоступные CGI.

Впрочем, разработка приложений для сервера в Kylix организована таким образом, что один тип приложений всегда можно без особого труда преобразовать в другой тип.

Принципы технологии CGI

Изначально протокол HTTP создавался для передачи по Сети HTML-документов и сопутствующей им графики, однако, Web-браузеры оказались настолько удобными, что Всемирная паутина очень быстро начала обрастать различными дополнительными функциями. Весьма заманчиво выглядит возможность использовать браузер клиента в качестве интерфейса для взаимодействия с удаленным приложением. Такое взаимодействие, разумеется, требует не только возможности посылать запросы на передачу определенных файлов, но и наличия механизма отправки команд и данных от клиента серверу. Для стандартизации механизма передачи команд от клиента к серверу и был разработан интерфейс CGI (Common Gateway Interface). В общих чертах работа CGI выглядит следующим образом: клиент загружает HTML-страницу, содержащую *форму*, и посылает серверу определенный в форме *запрос* (request), в котором указано *действие* (от англ. *action*, далее также будет использоваться термин "сервис") и список параметров. Действие представляет собой указатель на CGI-приложение, которое должно выполнить запрос. На-

пример, действие /cgi-bin/frenchwines указывает на приложение frenchwines. Для одного приложения можно определить несколько действий, которые идентифицируются как отдельные сетевые ресурсы: /cgi-bin/frenchwines, /cgi-bin/frenchwines/bordeaux, /cgi-bin/frenchwines/welcome — это разные действия (сервисы) приложения frenchwines. Список параметров представляет собой перечень имен параметров и их значений. Если HTML-форма использует для передачи параметров метод `GET`, список параметров можно увидеть в адресной строке браузера после идентификатора ресурса (он отделен от идентификатора символом `?`). Схематически это можно представить так:

http://localhost/cgi-bin/cgiapp/action1?param1=value1&param2=value2

Здесь action1 — один из сервисов приложения cgiapp, которому передаются параметры param1 и param2, со значениями value1 и value2 соответственно.

С точки зрения протокола HTTP CGI-запрос является командой на загрузку некоторого ресурса. Для передачи запроса могут использоваться HTTP-методы `POST`, `GET` или `HEAD`. В ответ на запрос сервер обычно посылает клиенту HTML-документ, который обычно представляет собой *динамическую страницу,* т.е. не хранится на диске в виде файла, а генерируется CGI-приложением. При этом CGI-приложение вовсе не обязано генерировать страницу целиком. Большинство динамических страниц содержит значительные по объему постоянные фрагменты. Для объединения статической и динамической информации на одной странице CGI-приложение может использовать шаблоны страниц (более подробно этот механизм будет рассмотрен в следующих главах), а также механизм включения страниц на стороне сервера (server side includes, SSI).

CGI-приложения представляют собой исполнимые модули, вызываемые Web-сервером. Схема взаимодействия между браузером (клиентом), сервером и CGI-модулем приводится на рис. 3.2. По своей структуре CGI-приложение является обычной консольной программой. Сервер передает данные CGI-приложению через переменные окружения и стандартный поток ввода, а CGI-приложение передает данные серверу через свой стандартный поток вывода. Сервер запускает новую копию приложения для обработки каждого запроса. По окончании обработки запроса CGI-приложение завершается. В связи с этим следует отметить важную особенность приложений CGI: эти приложения не сохраняют информацию в перерывах между транзакциями с клиентом.

Переменные окружения устанавливаются Web-сервером перед запуском CGI-модуля и передают приложению необходимую информацию о поступившем запросе. В табл. 3.1 приводится список стандартных переменных окружения CGI. Следует иметь в виду, что не все из этих переменных будут установлены для нашего приложения. Набор установленных переменных зависит от настроек клиента и сервера, а также от того, какой метод использовался при передаче запроса. Перед использованием очередной переменной окружения рекомендуется проверять, установлено ли соответствующее значение.

Таблица 3.1. *Переменные окружения для CGI-приложений*

Переменная	Описание
CONTENT_LENGTH	Длина в байтах блока информации, переданного через стандартный поток ввода
DOCUMENT_ROOT	Корневой каталог для HTML-документов Web-сервера
HTTP_REFERER	URL, содержащая форму, с которой поступил CGI-запрос
HTTP_USER_AGENT	Имя и номер версии удаленного приложения-клиента
PATH_INFO	Дополнительная информация о расположении CGI-сервиса. Может использоваться при предоставлении одним приложением нескольких сервисов
PATH_TRANSLATED	Дополнительная информация о расположении CGI-сервиса плюс DOCUMENT_ROOT
QUERY_STRING	Строка запроса. Определена при передаче запроса с помощью метода GET. Метод POST использует стандартный поток ввода для передачи запроса
REMOTE_ADDR	IP-адрес удаленного пользователя
REMOTE_HOST	Имя хоста удаленного пользователя
REQUEST_METHOD	Метод передачи запроса (POST, GET или HEAD)
SCRIPT_NAME	Путь к CGI-приложению в URL-пространстве имен
SERVER_NAME	Имя хоста или IP-адрес сервера
SERVER_PORT	Порт сервера

Как указано в таблице, метод GET передает строку запроса через переменные окружения, а метод POST — через стандартный поток ввода. Если нет уверенности, какой метод передачи данных использует Web-форма, требуется предусмотреть обработку обоих методов.

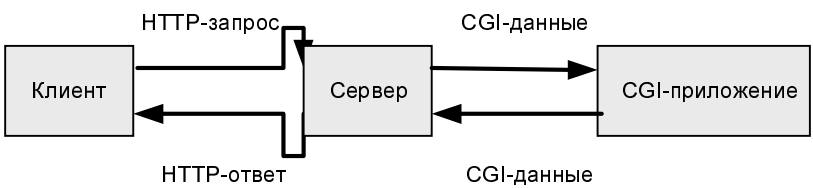


Рис 3.2. Взаимодействие между клиентом, сервером и CGI-приложением

Для того чтобы с помощью HTML-страницы можно было послать запрос, эта страница должна содержать либо гиперссылку на соответствующий сервис CGI-приложения с необходимыми параметрами, либо содержать HTML-форму. HTML-формы выделяются в тексте HTML-страницы при помощи тэгов `<FORM ...> ... </FORM>`. Более подробную информацию об HTML-формах можно найти в документации по языку разметки гипертекста HTML.

Создание разделяемых модулей для сервера Apache

Создать CGI-приложение для сервера Apache очень просто. Скомпилировав проект, для которого при создании был выбран пункт CGI Stand-Alone Executable, достаточно просто скомпилировать приложение в каталог cgi-bin сервера Apache. Web-сервер Apache позволяет также использовать разделяемые модули, обладающие более широкой функциональностью. Модули могут быть скомпилированы как часть ядра сервера Apache или же реализованы в виде разделяемых библиотек. Kylix позволяет создавать только модули в виде разделяемых библиотек. Для того чтобы сервер Apache мог подключать разделяемые библиотеки, необходимо пересобрать ядро сервера, установив опцию, разрешающую загрузку разделяемых модулей. Далее процесс конфигурации сервера и подключения разделяемых модулей будет описан для версий сервера 1.2.*x*—1.3.*x*, наиболее широко используемых в настоящее время. Для пересборки ядра нам потребуются исходные тексты сервера Apache (см. **www.apache.org**). Распаковав архив дистрибутива, в корневом каталоге дистрибутива надо создать файл config.status (листинг 3.1).

Листинг 3.1. Файл config.status

```
#!/bin/sh
LIBS="/usr/lib/libpthread.so" \
./configure \
"--with-layout=Apache" \
"--enable-module=so" \
"--enable-rule=SHARED_CORE" \
"$@"
```

Далее этому файлу следует присвоить статус исполняемого:

```
chmod 755 config.status
```

и запустить созданный скрипт:

```
./config.status
```

После этого выполняются обычные этапы сборки сервера.

Теперь можно перейти к написанию простейшего модуля. Для создания заготовки модуля Apache откроем Kylix Delphi IDE и в окне **New** выберем элемент **Web Server Application**. В открывшемся диалоговом окне выберем пункт **Apache shared module (DSO)**. Kylix создаст новый проект и в нем Web-модуль, являющийся аналогом главной формы приложения. Сохраним наш модуль под именем `apachedemo`. В инспекторе объектов щелкнем кнопкой мыши в поле свойства `Actions` объекта `WebModule1`. Перед нами откроется окно редактора **Editing WebModule1 Actions**. Создаем в этом окне новый элемент, переходим в инспектор объектов и назначаем свойству `Default` объекта `WebModule1.Actions[0]` значение `True`, а свойству `PathInfo` — значение `/`. Теперь переходим на страницу событий инспектора объектов и назначаем обработчик его событию `OnAction` объекта `WebModule1.Actions[0]` (листинг 3.2).

Листинг 3.2. Обработчик события `OnAction`

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<HTML><BODY><H2>Hello, Apache DSO↳
  World!</H2></BODY></HTML>';
end;
```

В этом листинге мы не вызываем метод `SendResponse`, так как не меняем значения `Handled`. Таким образом, запрос будет передан дальше для обработки стандартными средствами приложения. Теперь нам следует отредактировать исходный текст проекта. Откроем файл проекта и отредактируем текст так, чтобы он соответствовал листингу 3.3.

Листинг 3.3. Файл проекта apachedemo

```
library apachedemo;
uses
  WebBroker, HTTPD, ApacheApp, Main in 'Main.pas'
  {WebModule1: TWebModule};
exports
  apache_module name 'hello_module';
begin
  ModuleName := 'Hello_Module';
  ContentType := 'hello-handler';
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

Значения переменных `ModuleName` и `ContentType` необходимы серверу Apache для идентификации разделяемого модуля.

В результате компиляции проекта у нас получится файл libapachedemo.so. Скопируем этот файл в каталог разделяемых модулей сервера Apache (если сервер был установлен в каталог /usr/local/apache/, каталогом разделяемых модулей будет/usr/local/apache/libexec) и добавим в файл httpd.conf фрагмент, представленный в листинге 3.4.

Листинг 3.4. Дополнение файла httpd.conf

```
LoadModule hello_module libexec/libHelloModule.so
<Location /hello>
SetHandler hello-handler
</Location>
```

После директивы `LoadModule` следует имя модуля (указанное в директиве `exports` библиотеки) и путь к файлу, в котором размещен модуль. Команда `Location` позволяет задать URL, соответствующую обработчику `hello-handler`. Для того чтобы проверить работу разделяемого модуля, надо перезапустить Web-сервер и в адресной строке браузера набрать: **http://localhost/hello**.

Глава 4

Hello, Internet World

В этой главе мы рассмотрим простейшие интернет-приложения — приложение-клиент, использующее сокеты, CGI-приложение и простейший сервер. А также познакомимся с основными технологиями разработки приложений для Интернета в Kylix — Internet Direct, WebBroker, WebSnap и WebServices. Более углубленное рассмотрение всех этих технологий приведено в главах 5, 7—9. Данная глава должна позволить программисту, не знакомому со средствами разработки интернет-приложений от Borland, разобраться в том, какие именно технологии программирования наилучшим образом подходят для решения стоящих перед ним задач.

Сокеты — это просто

В разделе, посвященном взаимодействию между процессами *(см. гл. 1)*, мы уже познакомились с сокетами в файловом пространстве имен. Однако, основное назначение сокетов — передавать данные по сети. В этом разделе мы рассмотрим сетевые сокеты, являющиеся основой интерфейса сетевого программирования в ОС Linux. Сетевые сокеты можно рассматривать как концевые точки, образующие сетевое соединение. Для тех, кто уже писал программы с использованием сокетов для ОС Windows, отметим, что программирование на уровне сокетов в ОС Linux выполняется гораздо проще.

Конечно, при работе в Kylix обращаться непосредственно к сетевым функциям ОС Linux нет необходимости. Функциональность сокетов инкапсулирована в классе `TSocket` и его потомках, позволяющих работать с сетевыми протоколами на высоком уровне. Тем не менее программирование на уровне сокетов наглядно демонстрирует некоторые принципы, лежащие в основе многих сетевых компонентов, поэтому для начинающего интернет-программиста будет полезно рассмотреть структуру простейшего сетевого приложения. Далее мы напишем программу-клиент (листинг 4.1), посылающую запрос `GET` HTTP-серверу.

Листинг 4.1. Программа-клиент, использующая сокеты

```
//---------------------------------------------------------------------------


#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#pragma hdrstop

//---------------------------------------------------------------------------

int main()
{
  int sock;
  struct sockaddr_in server;
  struct hostent * host;
  char * addr = "localhost";
  char * HTTP_MSG = "HEAD / HTTP/1.0\n\n";
  char buf[1024];
  int len;
  sock = socket(PF_INET, SOCK_STREAM, 0);
  host = gethostbyname(addr);
  server.sin_family = AF_INET;
  server.sin_port = htons(80);
  server.sin_addr  = *(struct in_addr *) host->h_addr;
  connect(sock, (struct sockaddr *) &server, sizeof(server));
  write(sock, HTTP_MSG, strlen(HTTP_MSG) + 1);
  len = read(sock, buf, 1023);
  buf[len] = 0;
  printf("Ответ сервера:\n %s", buf);
  shutdown(sock, 2);
  return 0;
}
//---------------------------------------------------------------------------
```

Новый сетевой сокет создается при помощи функции `socket`, первым параметром которой является семейство протоколов (в нашем случае — `AF_INET`), а вторым — тип сокета (потоковый или датаграммный).

Установка связи с узлом сети осуществляется функцией `connect`, которой необходимо передать дескриптор сокета, возвращенный функцией `socket`, и

указатель на структуру `sockaddr`, идентифицирующую узел сети. Поскольку мы работаем с протоколами Интернета, нам следует использовать структуру `sockaddr_in`, тип которой будет преобразован в тип `sockaddr`. Узел сети идентифицируется сетевым именем (в нашем случае это адрес localhost — имя локального узла), а сервис — номером порта (мы используем порт 80 — стандартный порт сервера HTTP). Для получения адреса узла в формате, пригодном для использования сетевыми функциями, служит функция `gethostbyname`. Возможности этой функции гораздо шире, чем может показаться из приведенного листинга. Функция `gethostbyname` сначала пытается определить IP-адрес узла с указанным именем, используя файл hosts локальной системы, а при необходимости обратится к интернет-службе DNS, и вернет значение IP-адреса узла, пригодное для использования другими сетевыми функциями.

Заполняя структуру `sockaddr_in`, мы использовали функцию `htons`. Эта функция конвертирует числовые значения из форматов, принятых на данной машине, в форматы, принятые в Интернете.

После того, как соединение установлено, отправку и прием данных можно осуществлять при помощи стандартных функций `write` и `read`, используя дескриптор сокета. В листинге 4.2 приводится результат выполнения программы.

Листинг 4.2. Результат отправки запроса HTTP-серверу

```
Ответ сервера:
HTTP/1.1 200 OK
Date: Wed, 23 Oct 2002 10:49:34 GMT
Server: Apache-AdvancedExtranetServer/1.3.23
(Mandrake Linux/4mdk) mod_ssl/2.8.7 OpenSSL/0.9.6c PHP/4.1.2
Connection: close
Content-Type: text/html
```

Закрывается сокет при помощи функции `suhtdown`. Второй параметр этой функции указывает, какие именно операции с сокетом следует прекратить:

- 0 — чтение данных;
- 1 — запись данных;
- 2 — чтение и запись.

Различные варианты закрытия сокета позволяют выполнить "вежливое закрытие" (graceful shutdown). Манипулируя этими значениями, можно вовремя проинформировать удаленную сторону о том, что сокет закрывается, и разрыв соединения не станет неожиданностью для удаленного приложения.

Пишем программы для сервера: простое CGI-приложение на C++

Kylix 3 предоставляет в распоряжение программиста многочисленные средства, ускоряющие и упрощающие разработку CGI-приложений, но в этом разделе мы напишем CGI-приложение, используя лишь стандартные средства языка C++. Это приложение позволит лучше понять общие принципы разработки CGI-приложений. Перед тем как приступить к анализу простого CGI-приложения, рекомендуется ознакомиться с разделом "Принципы технологии CGI" в главе 3.

Прежде чем рассмотреть демонстрационное CGI-приложение, напишем HTML-страницу, содержащую форму для отправки CGI-запросов. В листинге 4.3 приводится текст страницы testcgi.html.

Листинг 4.3. Страница testcgi.html

```
<html>
  <head>
    <title>Test CGI Page</title>
    <meta content="">
  </head>
  <body>
  <H2>CGI-форма</H2>
  <P>Напечатайте строку и нажмите "submit"</P>
  <form action=http://first.intranet/cgi-bin/simplecgi method=GET>
  <P>Эта форма использует метод GET</P>
  <P><input type=text size=20 name=text>
  <input type=submit name=submit>
  </form></P>
  <P><form action=http://first.intranet/cgi-bin/simplecgi method=POST>
  <P>Эта форма использует метод POST</P>
  <P><input type=text size=20 name=text>
  <input type=submit name=submit>
  </form></P>
  </body>
</html>
```

В этой странице две формы, посылающие CGI-запрос приложению simplecgi. Одна форма использует метод `GET`, другая — `POST`. Строка **http://first.intranet** указывает путь к серверу. Заменим ее на имя соответствующего хоста в нашей сети, или на значение **http://localhost**.

Скопируем эту страницу в каталог, в котором хранятся Web-страницы нашего сервера.

Теперь напишем само приложение simplecgi. Для этого нам понадобится создать заготовку консольного проекта C++. Исходный текст главного модуля приводится в листинге 4.4.

Листинг 4.4. Приложение simplecgi

```
//---------------------------------------------------------------------------

#include <stdlib.h>
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
  char * Method, * Val;
  Method = getenv("REQUEST_METHOD");
  if (Method == NULL)
  {
    cout << "Ошибка, вы пытаетесь запустить CGI-приложение\n";
    return 1;
  }
  cout << "Content-Type: text/html\n\n";
  cout << "<HTML><HEAD><META HTTP-EQUIV=\"Content-Type\"↳
  CONTENT=\"text/html; charset=koi8-r\">";
  cout << "<TITLE>Простое CGI-приложение</TITLE></HEAD><BODY>";
  cout << "<H2>Эта страница создана CGI-приложением<//H2>";
  cout << "<P>Вы использовали метод " << Method << "</P>";
  if ((Val = getenv("HTTP_USER_AGENT")) != NULL)
  cout << "<P>Клиент: " << Val << "</P>";
  if ((Val = getenv("HTTP_REFERER")) != NULL)
  cout << "<P>URL запроса: " << Val << "</P>";
  if (strcmp(Method, "GET") == 0)
  {
    if ((Val = getenv("QUERY_STRING")) != NULL)
```

```
    cout << "<P>Строка запроса: " << Val << "</P>";
  }
  else
  {
    int clen;
    char * Query;
    if ((Val = getenv("CONTENT_LENGTH")) != NULL)
    {
      cout << "<P>Длина контента: " << Val << " байт</P>";
      clen = atoi(Val);
      Query = (void*) malloc(clen);
      cin >> Query;
      cout << "<P>Строка запроса: </P>" << Query;
      free(Query);
    }
  }
  cout << "</BODY></HTML>";
  return 0;
}
//---------------------------------------------------------------------------
```

Наша программа получает значительную часть данных о запросе из переменных окружения. Получить значение переменных окружения можно с помощью функции `getenv`. Эта функция возвращает либо указатель на строку с концевым нулем, содержащую значение запрошенной переменной, либо значение `NULL`, если соответствующая переменная не установлена.

В начале программы мы проверяем, установлено ли значение переменной окружения `REQUEST_METHOD`. Web-сервер *всегда* устанавливает эту переменную, но наша программа simplecgi — обычное консольное приложение, и нет никакой гарантии, что кто-то не попытается выполнить его "просто так". Проверка переменной `REQUEST_METHOD` защищает программу от неправильного использования. Если эта переменная не установлена, значит, приложение запустил пользователь, а не Web-сервер, и мы выводим сообщение об ошибке. Если метод запроса указан, мы начинаем генерацию HTML-страницы. Обратите внимание на два символа "новая строка" после первой строки контента. Это правило является частью протокола HTTP.

Если CGI-запрос передан при помощи метода `POST`, нам следует определить длину переданного блока информации, выделить блок памяти соответствующего размера и прочесть данные из потока ввода. В Delphi Language все это можно было бы сделать даже проще — при помощи переменной типа `String` и процедуры `ReadLn`.

Скомпилируем приложение simplecgi и скопируем полученный двоичный модуль в каталог cgi-bin нашего сервера. При этом нам, возможно, понадобится изменить владельца и группу для файла simplecgi в соответствии с настройками сервера.

В строке адреса нашего браузера наберем: **http://localhost/testcgi.html**. В окне браузера откроется страница с формами для передачи данных CGI-приложению. Наберем какой-нибудь текст в строке ввода одной из форм и нажмем кнопку **Отправить** (Submit). Если все сделано правильно, в окне браузера должно появиться нечто похожее на рис. 4.1.

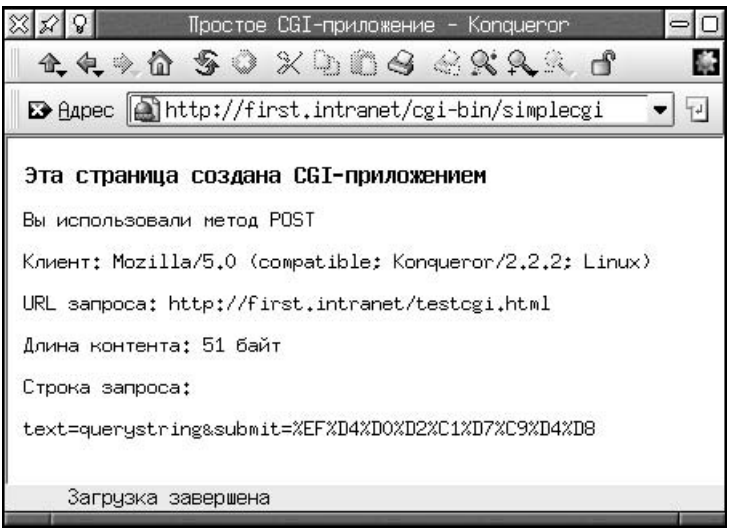


Рис 4.1. Динамическая HTML-страница, созданная приложением simplecgi

Конечно, генерация всей HTML-страницы внутри приложения — не очень удобный способ, особенно если учесть, что значительная часть любой CGI-страницы является статической, т. е. одинаковой для всех CGI-запросов. Существует ряд методов, упрощающих генерацию CGI-страниц. Прежде всего можно воспользоваться уже упоминавшейся технологией SSI. В статической HTML-странице, размещенной на сервере, следует указать специальные тэги, которые перед отправкой страницы клиенту будут заменяться содержимым указанного в них ресурса. Основной сферой применения этих SSI-тэгов как раз и является объединение статических фрагментов HTML-страниц и результатов выполнения CGI-приложений. SSI-тэг имеет вид

```
<!--#include ссылка_на_ресурс -->
```

Например, для того чтобы включить в страницу, отправляемую клиенту, результат выполнения приложением simplecgi CGI-запроса `query_string`, следует включить в страницу тэг

```
<!--#include virtual="/cgi-bin/simplecgi?query_string" -->
```

Однако этот метод не всегда применим, так как при его использовании трудно организовать обработку произвольных запросов. Лучше всего данный метод подходит для вставки в большие статические страницы небольших динамических фрагментов с постоянной формой запроса (например, число посетителей сервера в данный момент, рейтинг ресурса и т. п.).

Программирование сетевых демонов

В этом разделе мы познакомимся с основами программирования сетевых демонов (и, наконец-то, ответим на вопрос, что это такое). То, о чем пойдет здесь речь, вряд ли понадобится прикладному программисту. Однако затронутые в этом разделе вопросы позволят нам лучше понять особенности работы сетевой подсистемы ОС Linux, да и особенности программирования Linux вообще.

Демон (daemon, demon) — это программа, выполняющаяся в фоновом режиме и настроенная на обработку определенного типа событий ОС. После запуска демон находится в состоянии ожидания события. Когда в системе происходит соответствующее событие, демон активизируется и выполняет необходимые действия по его обработке. Важной особенностью демонов является возможность параллельной обработки нескольких событий. Демоны бывают не только сетевыми. Например, печатью в ОС Linux управляет демон печати, а записью сообщений в log-файлы — log-демон (`logd`).

Сетевой демон находится в состоянии ожидания запроса на соединение. Когда такой запрос поступает, демон либо устанавливает соединение и выполняет обработку запроса, либо отказывается установить соединение. Следует отметить, что программа-сервер Интернета вовсе не обязательно должна быть демоном. В следующих разделах мы напишем серверы, обладающие графическим интерфейсом. Однако если нужно, чтобы сервер запускался вместе с операционной системой и выполнялся вне зависимости от того, кто из пользователей работает в системе в данный момент, реализация сервера в форме демона представляется наилучшим вариантом.

Демон - это консольное приложение, выполняющееся в фоновом режиме. Фоновым режимом (background mode) называется такой режим работы программы, в котором приложение не взаимодействует с терминалом. После запуска демон должен переходить в фоновый режим автоматически.

Как уже было отмечено, демон не должен блокировать поступающие события, т. е. программа должна быть способна обрабатывать несколько событий одновременно. Решить эту задачу можно либо при помощи многопоточности, либо, как это делается в "классических" демонах, при помощи уже встречавшейся нам функции `fork`.

Примечание

Функция `fork` играет чрезвычайно важную роль в работе демонов. Именно поэтому демон, являющийся эмблемой FreeBSD, держит в лапах вилы.

Как же выполняется перевод демона в фоновый режим? Самый простой способ сделать это — создать копию процесса, завершить родительский процесс, а в дочернем процессе закрыть поток ввода и перенаправить потоки вывода на устройство /dev/null. Просто закрыть потоки вывода нельзя, так как многие функции системных библиотек, которые понадобятся нашему демону, используют эти потоки для вывода различной информации. Закрытие потоков вывода привело бы к ошибкам при вызове таких функций.

Таким образом, приложение-демон перестает взаимодействовать с терминалом. Но это еще не все. Для того, чтобы демон стал окончательно независим от запустившего его терминала, следует создать новую сессию. Процесс переводится в новую сессию при помощи функции `setsid`. Как и функция `fork`, функция `setsid` возвращает новый идентификатор процесса.

Осталось ответить на еще один важный вопрос: как управлять приложением, выполняющимся в фоновом режиме? Остановить любой Linux-процесс можно с помощью команды `kill`, однако для "цивилизованного" управления процессами это средство явно не подходит. Кроме того, команде `kill` требуется идентификатор процесса, а найти его не так-то просто (для поиска идентификатора конкретного процесса лучше всего использовать сочетание команд `ps` и `grep`). Обычно программы-демоны поставляются вместе со специальными утилитами управления. Например, упоминавшийся выше сервер Apache (демон `httpd`) поставляется вместе с программой apachectl, позволяющей выполнять такие операции, как запуск, останов и перезапуск демона.

Утилита, управляющая работой демона, осуществляет управление при помощи сигналов. Но для того, чтобы послать другому процессу сигнал, нужно опять-таки получить идентификатор этого процесса. Для передачи идентификатора процесса утилите управления можно воспользоваться механизмами межпроцессного взаимодействия, однако традиционные программы-демоны идут другим путем. Обычно демон записывает идентификатор своего процесса в специальный файл на диске. Имя файла совпадает с именем программы-демона, а в качестве расширения служит `.pid`. Например, файл идентификатора сервера Apache имеет имя httpd.pid. Файлы идентификаторов хранятся в специальном каталоге (обычно -/var/run/).

Поскольку управление демоном осуществляется посредством сигналов, в приложении-демоне необходимо организовать корректную обработку сигналов ОС Linux. Демоны должны обрабатывать, как минимум, два сигнала — `SIGTERM` и `SIGHUP`. Первый сигнал используется для остановки процесса-демона, реакцией на второй сигнал обычно является перезапуск процесса.

Очевидно, что демон не может непосредственно информировать пользователя о различных событиях, происходящих во время его работы. Вместо этого все, что происходит во время работы демона, записывается в специальные файлы журналов (log-файлы). Например, сервер Apache ведет несколько log-файлов — журнал соединений и журнал ошибок. Log-файлы настолько важны для операционной системы, что она предоставляет специальные средства для ведения системных log-файлов. Системными log-файлами управляет специальный демон — `logd`. Системные библиотеки предоставляют набор функций для работы с этим типом log-файлов.

В качестве практического примера рассмотрим простейший сервер протокола HTTP, выполненный в форме демона. Для обработки запросов демон использует класс TIdHTTPServer из набора компонентов Indy. Исходный текст главного модуля программы-демона приводится в листинге 4.5, а исходный текст вспомогательного модуля `WebServ` — в листинге 4.6.

Листинг 4.5. Исходный текст программы dhttpd

```
program dhttpd;

{$APPTYPE CONSOLE}

uses
  Libc, SysUtils, WebServ;

var
  F : Text;
  SigSet : __sigset_t;
  SigNum : Integer;
  pid : Integer;

// Процедура обработки сигналов

procedure TranslateSig(sig : Integer); cdecl;
begin
  SigNum := sig;
end;

begin

// Обработка параметров командной строки
```

```
  if ParamCount > 0 then
  begin
    if ParamCount = 1 then
    begin
      SigNum := 0;
      if ParamStr(1) = '-stop' then SigNum := SIGTERM else
      if ParamStr(1) = '-restart' then SigNum := SIGHUP;
      if SigNum <> 0 then
      begin
        if FileExists('/var/run/dhttpd.pid') then
        begin
          AssignFile(F, '/var/run/dhttpd.pid');
          Reset(F);
          Read(F, pid);
          CloseFile(F);
          if kill(pid, SigNum) = 0 then
          Halt(0);
        end;
        WriteLn('dhttpd : Невозможно выполнить ' + ParamStr(1) +
       '. Может быть, демон не запущен?');
        Halt(0);
      end;
    end;
    WriteLn('Используйте dhttpd -stop для остановки демона или↳
    dhttpd -restart для перезапуска.');
    Halt(0);
  end;

// Инициализация демона

  pid := fork;
  if pid = -1 then // ошибка fork
  begin
    perror(PChar('fork 1: '));
    Halt(1);
  end else
  if pid <> 0 then Halt(0); // родительский процесс завершается
  pid := setsid;
```

```
if pid = -1  then  // ошибка setsid
begin
  perror(PChar('setsid: '));
  Halt(1);
end;
System.Close(Input);    // Закрываем stdin
AssignFile(Output, '/dev/null');
Rewrite(Output);
AssignFile(ErrOutput, '/dev/null');
Rewrite(ErrOutput);

// Теперь процесс выполняется в режиме демона

WebServer := TWebServer.Create;  // создаем объект-сервер
// Записываем идентификатор процесса
AssignFile(F, '/var/run/dhttpd.pid');
Rewrite(F);
pid := getpid;
WriteLn(F, pid);
System.Close(F);
OpenLog;
WriteLog('Демон запущен');
signal(SIGTERM, @TranslateSig);
signal(SIGHUP, @TranslateSig);
sigfillset(SigSet);
sigdelset(SigSet, SIGTERM);
sigdelset(SigSet, SIGHUP);
SigNum := 0;
while SigNum <> SIGTERM do
begin
  sigsuspend(SigSet);
  WebServer.Free;
  WriteLog('Демон завершил работу');
  CloseLog;
  if SigNum = SIGHUP then    // Перезапуск
  begin
    pid := fork;
    if pid = -1 then Halt(1);
```

```
      if pid <> 0 then Break;  // Прежний процесс завершается
      // Теперь мы в новом дочернем процессе
      WebServer := TWebServer.Create;
      AssignFile(F, '/var/run/dhttpd.pid');
      Rewrite(F);
      pid := getpid;
      WriteLn(F, pid);
      System.Close(F);
      OpenLog;
      WriteLog('Демон успешно перезапущен');
    end;
  end;
end.
```

Особенностью нашего демона является то, что для управления демоном используется сама же программа `dhttpd`. При запуске программа проверяет наличие параметров командной строки. Если командная строка пуста, программа запускается в режиме демона. Если в командной строке есть параметры, программа читает идентификатор процесса-демона из pid-файла и посылает ему соответствующий сигнал. При этом предполагается, что демон уже запущен.

Для работы с log-файлами программа использует функции `OpenLog`, `WriteLog` и `CloseLog`, определенные в модуле `WebServ`.

Важная часть демона — цикл обработки сигналов. Для того, чтобы уменьшить нагрузку на систему, мы используем функцию `sigsuspend`. В качестве единственного параметра этой функции передается маска сигналов — переменная типа `__sigset_t`. Функция `sigsuspend` приостанавливает выполнение вызвавшего ее потока до тех пор, пока не поступит один из сигналов, не включенных в маску. В этом случае, если обработчик сигнала не завершает процесс, функция `sigsuspend` возвращает управление. С помощью функции `sigfillset` мы создаем полную маску сигналов, а с помощью функции `sigdelset` удаляем из маски сигналы `SIGTERM` и `SIGHUP`, которые наш демон должен обрабатывать.

Приостановка процесса функцией `sigsuspend` не помешает демону обрабатывать сетевые запросы, так как занимающийся этим класс Indy использует отдельные потоки. Использование потоков представляется более эффективным методом параллельной обработки процессов, нежели "клонирование" основного процесса при помощи `fork` для обработки каждого запроса.

Сетевая часть демона `httpd` реализована в модуле `WebServ`, текст которого приводится в листинге 4.6.

Листинг 4.6. Модуль WebServ

```
unit WebServ;

interface

uses
  IdBaseComponent, IdComponent, IdTCPServer, IdHTTPServer, SysUtils;

{ Процедуры для работы с log-файлами. Мы располагаем их
  здесь, чтобы оба модуля имели к ним доступ. }

var
  LOGF : Text;

procedure OpenLog;
begin
  System.Assign(LOGF, '/var/log/dhttpd.log');
  System.Append(LOGF);
end;

procedure CloseLog;
begin
  System.Close(LOGF);
end;

procedure WriteLog(const Msg : String);
begin
  WriteLn(LOGF, DateTimeToStr(DateTime), ': ', Msg);
end;

type

  TWebServer = class(TObject)
    IndyHTTP : TIdHTTPServer;
  private
    procedure HTTPCommandGet(AThread : TIdPeerThread;
    RequestInfo : TIdHTTPRequestInfo; ResponseInfo :
TIdHTTPResponseInfo);
```

```
  public
    constructor Create;
    destructor  Destroy; override;
  end;

const
  HTTPDocsDir = '/var/www/html'; // Корневой каталог документов сервера

var
  WebServer : TWebServer;

implementation

constructor TWebServer.Create;
begin
  inherited Create;
  IndyHTTP := TIDHTTPServer.Create(nil);
  with IndyHTTP do
  begin
    ServerSoftware := 'DAEMONstration Web server.';
    OnCommandGet := HTTPCommandGet;
    TerminateWaitTime := 5000;
    DefaultPort := 8800; // Порт сервера
    // Активируем сервер
    Active  := True;
  end;
end;

destructor TWebServer.Destroy;
begin
  IndyHTTP.Free;
  inherited Destroy;
end;

procedure TWebServer.HTTPCommandGet;
var
  FName, LogStr : String;
begin
```

```
  FName := HTTPDocsDir + RequestInfo.Document;
  LogStr := Format('Клиент %s запросил ресурс %s; ',
    [@RequestInfo.RemoteIP[1], @FName[1]]);
  if FileExists(FName) then
  begin
    // Возвращаем контент
    IndyHTTP.ServeFile(AThread, ResponseInfo, FName);
    LogStr := LogStr + 'OK';
  end else
  begin
    // Запрошенный ресурс не найден
    ResponseInfo.ResponseNo := 404;
    ResponseInfo.ResponseText := 'Запрошенный ресурс не найден';
    ResponseInfo.ContentText := 'Ошибка 404 — ' +
      ResponseInfo.ResponseText;
    ResponseInfo.WriteHeader;
    ResponseInfo.WriteContent;
    LogStr := LogStr + 'Ошибка: ресурс не найден.';
  end;
  WriteLog(LogStr);
end;
```

Для того, чтобы не конфликтовать с Web-сервером Apache, наш Web-сервер слушает порт 8800. Если в системе этот порт уже занят каким-либо сервисом, можно использовать другое значение. Конечно, сервер `dhttpd` довольно примитивен. Обрабатывается только HTTP-запрос `GET` и не производится перенаправление запроса в случае передачи неполной ссылки. Однако нельзя не отметить простоту использования компонентов Internet Direct. С этими компонентами мы и познакомимся подробнее в следующем разделе. Для того, чтобы запустить наш демон, необходимо дать команду `dhttpd`. Команда `dhttpd -stop` останавливает демон, а команда `dhttpd -restart` перезапускает его. Для того, чтобы демон стал компонентом системы, т. е. запускался автоматически при загрузке ОС, следует выполнить специальную настройку системы. Вопросы настройки ОС Linux выходят за рамки этой книги.

Знакомство с компонентами Internet Direct

Начиная с Delphi 4, компания Borland использует компоненты, предназначенные для быстрой разработки независимых интернет-приложений, созданные сторонними разработчиками. В последних версиях своих продуктов Borland использует компоненты Internet Direct (Indy) от компании Nevrona, и этот выбор следует признать весьма удачным. Прежде всего, Internet Direct — это от-

крытый набор компонентов с доступными исходными текстами, предназначенный для использования в средствах разработки Borland на платформах Windows и Linux. Набор компонентов, входящих в состав Indy чрезвычайно широк. Кроме готовых компонентов-клиентов и серверов, реализующих функциональность соответственно клиентов и серверов наиболее распространенных протоколов Интернета, Indy включает ряд вспомогательных компонентов, облегчающих решение многих распространенных задач, с которыми сталкиваются интернет-программисты. В пакет Indy входит также множество демонстрационных примеров (к сожалению, не все из этих примеров адаптированы для Kylix). Сами же компоненты Internet Direct работают в середе Linux с отменной надежностью. Наличие версий пакета для обеих платформ делает приложения, написанные с помощью Indy, переносимыми.

Отличительной особенностью компонентов Indy является использование блокирующих операций при работе с сокетами. Напомним, что блокирующими называются вызовы, приостанавливающие выполнение вызвавшего их потока до окончания соответствующей операции. Использование блокирующих операций упрощает разработку сетевой подсистемы и облегчает перенос компонентов на Unix-платформы.

В принципе использование блокирующих операций в приложениях с графическим интерфейсом, а также в приложениях, предназначенных для параллельной обработки множества запросов, считается неудобным. Однако пакет Indy предлагает сразу два решения этой проблемы. Во-первых, компоненты Indy чрезвычайно широко используют многопоточность. В предыдущем примере (листинг 4.6) нам не нужно было заботиться о реализации механизма обработки запросов. Класс `TIDHTTPServer` все уже сделал за нас. При работе с компонентами Indy создание нижележащих потоков и управление ими происходит незаметно для программиста (хотя при желании можно получить доступ к потокам). К преимуществам использования отдельных потоков для обработки соединений следует отнести также и то, что потоки позволяют организовать лучшее разделение ресурсов системы между задачами, чем это было бы при асинхронной обработке запросов одним процессом.

Принцип работы многопоточного компонента-сервера показан на рис. 4.2. В ответ на входящий запрос компонент-сервер создает поток, в котором выполняется обработка запроса. По мере необходимости поток-обработчик запроса вызывает события компонента-сервера. При этом, естественно, реализован механизм, исключающий одновременный вызов событий несколькими потоками. Обработчик события `OnCommandGET` в листинге 4.6 получает ссылку на объект, инкапсулирующий поток, обрабатывающий запрос, и объекты `RequestInfo` и `ResponceInfo`, позволяющие получить и передать дополнительные данные о запросе.

Компоненты-клиенты далеко не всегда нуждаются в том, чтобы обрабатывать несколько соединений одновременно. В этом случае работа с компонентом может выполняться по следующей схеме (листинг 4.7).

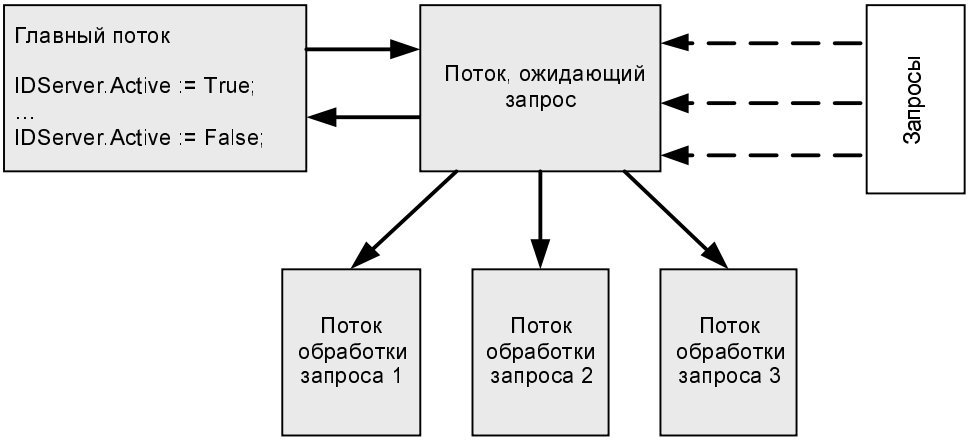


Рис. 4.2. Многопоточная модель компонента Indy

Листинг 4.7. Принцип работы с компонентом-клиентом Indy

```
IndyClient.Connect;
try
// Отправка запроса на сервер и получение данных
finally
  IndyClient.Disconnect;
end;
```

Как же добиться того, чтобы использование компонента-клиента не "замораживало" пользовательский интерфейс приложения? Оказывается, это очень просто. Для решения этой задачи в Indy реализован класс `TIdAntiFreeze` (компонент `IdAntiFreeze`). Для того чтобы приложение могло работать с блокирующими функциями сетевого интерфейса и при этом продолжало реагировать на события графической среды, достаточно разместить этот компонент в главной форме приложения. Компонент `IdAntiFreeze` прерывает выполнение блокирующих операций и периодически вызывает метод `ProcessMesages`. Программист же может работать с функциями в блокирующем режиме.

В качестве практического примера использования компонентов-клиентов рассмотрим фрагменты исходных текстов HTTP-клиента, написанных с помощью Indy. Приложение-клиент представлено на рис. 4.3.

Приводить полные исходные тексты приложения-клиента мы не будем. Сосредоточим внимание лишь на отдельных моментах. Основой нашего приложения является компонент `IdHTTP`. Послать запрос серверу и получить возвращенные данные с помощью компонента `IdHTTP` очень просто. Для

отправки запроса можно использовать один из двух методов `Get` класса `TIdHTTP`. Оба метода получают в качестве параметра строку, содержащую URL запроса. Один из методов возвращает контент в виде строки (тип `String`):

```
Content := IdHTTP1.Get('http://localhost/');
```

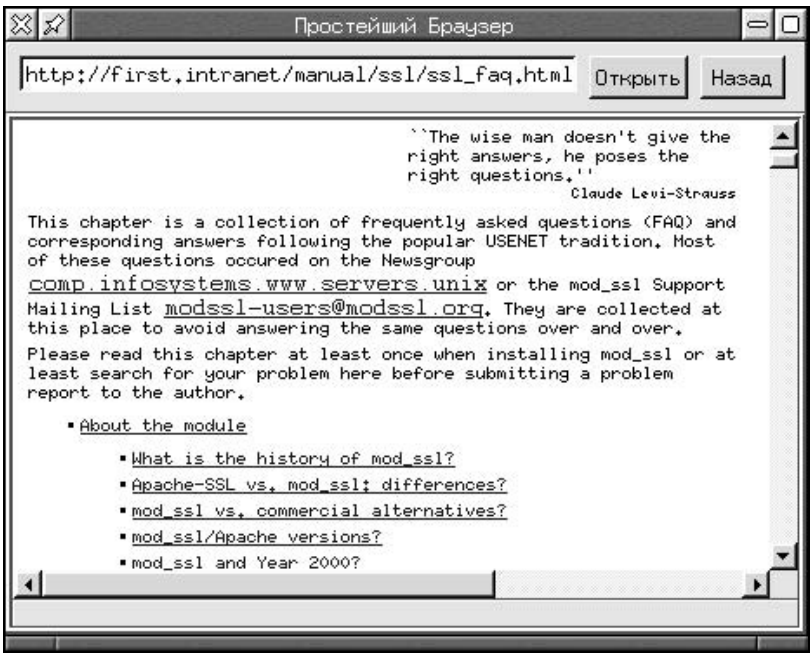


Рис. 4.3. Приложение-клиент HTTP

Другой вариант метода `Get` позволяет сохранить контент с помощью класса-наследника `TStream`. В листинге 4.8 приводится исходный текст процедуры `SaveContent`, сохраняющей данные, возвращенные сервером, в файле на диске.

Листинг 4.8. Процедура `SaveContent`

```
procedure SaveContent(const URL, FileName : String);
var
  FS : TFileStream;
begin
try
    FS := TFileStream.Create(FileName, fmCreate);
    IdHTTP1.Get(URL, FS);
```

```
    FS.Free;
  except
    on E : EIdException do
    begin
      FS.Free;
      ShowMessage(E.Message);
    end;
    on E: Exception do ShowMessage(E.Message);
  end;
end;
```

Как и другие методы запросов Indy-клиентов, метод `Get` возвращает управление только после того, как соответствующая операция будет завершена. Функция `SaveContent` представляет также пример обработки исключений Indy. Обработка исключений при использовании компонентов Indy играет чрезвычайно важную роль. Можно даже сказать, что обработка исключений является необходимым условием нормальной работы Indy-программы. Дело в том, что именно при помощи исключений компоненты Indy информируют программу об ошибках и нештатных ситуациях, а такие ситуации в сетевых приложениях возникают чаще, чем в любых других. Indy генерирует исключения при разрыве соединения с Интернетом, возвращении удаленным сервером кода ошибки и в других подобных случаях.

В листинге 4.8 мы использовали класс `EIdException`, базовый класс исключений Indy. Именно этот класс следует использовать для обработки исключений Indy, если конкретный тип исключения не имеет значения. Класс `EIdException` является потомком класса `Exception`, поэтому, если в обработчике исключения задействован класс `Exception`, обработку `EIdException` следует включить до выполнения операций с классом `Exception`.

Класс `EIdException` и его потомки, реализующие определенные типы исключений, определены в модуле `IdException`. К наиболее распространенным исключениям Indy относятся:

- `EIdInvalidSocket` — исключение, возникающее при внезапном разрыве соединения;
- `EIdProtocolReplyError` — исключение, возникающее в результате ошибки протокола передачи данных;
- `EIdResponseError` — ошибка в ответе сервера;
- `EIdConnClosedGracefully` — исключение, возникающее при завершении соединения. Если это исключение вызвано компонентом-сервером, скорее всего его причиной стал намеренный разрыв соединения со стороны клиента, и тогда его не следует рассматривать как ошибку (хотя обрабатывать все равно желательно). Если же исключение вызвано компонен-

том-клиентом, это значит, что сервер не смог выполнить запрос, и пользователя следует проинформировать об ошибке.

Технология WebBroker

В начале этой главы уже говорилось о методах интеграции статического и динамического HTML-контента. Этот раздел посвящен технологии WebBroker, которая предоставляет еще один метод интеграции на основе компонентов-генераторов контента (page producers) и шаблонов, а также ряд других полезных методов. Фактически технология WebBroker является основой разработки приложений для сервера в среде Kylix. Эта технология позволяет осуществлять обработку CGI-запросов традиционным способом для объектной модели среды Kylix .

Компоненты-генераторы создают динамическую HTML-страницу на основе шаблона, содержащего статическую часть страницы и специальные тэги. В процессе создания компонентом динамической страницы эти тэги заменяются значениями, переданными CGI-приложением, после чего страница отправляется клиенту. Технология WebBroker будет рассмотрена подробно в главе 7, здесь же мы ограничимся простым приложением.

Запустим среду Kylix C++, откроем диалоговое окно **New Items** и на странице **New** выберем пункт **Web Server Application**. В открывшемся окне укажем тип приложения — **CGI Stand-alone executable**.

На экране появится форма `WebModule1`. Эта форма играет в разработке приложений WebBroker ту же роль, что и главная форма в разработке обычных приложений. Сохраним новый проект под именем cgidemo.

Поместим в форму компонент `PageProducer` (страница **Internet** палитры компонентов). Компонент `PageProducer` — самый простой из компонентов-генераторов контента.

В разделе, посвященном описанию CGI, говорилось, что в поступающих CGI-приложению запросах, кроме параметров запроса, указывается действие (action). Каждое действие можно рассматривать как отдельный сервис, предоставляемый данным CGI-приложением. Таким образом, одно CGI-приложение может предоставлять несколько сервисов. С точки зрения протокола HTTP указание действий-сервисов выглядит, как указание дополнительного пути. Например, различные сервисы приложения cgidemo могут идентифицироваться как /cgi-bin/cgidemo/, /cgi-bin/cgidemo/action1/, /cgi-bin/cgidemo/otheraction/ и т. п.

Для того, чтобы наше CGI-приложение работало, нам нужно создать хотя бы один сервис. Технология WebBroker позволяет автоматизировать многие этапы создания CGI-сервиса и сводит к минимуму количество строк кода, которые придется набирать вручную. Форма главного модуля приложения

WebBroker имеет свойство Actions, которое представляет собой список объектов-сервисов данного приложения.

В инспекторе объектов выбираем объект WebModule1 и щелкаем кнопкой мыши в поле свойства Actions. При этом открывается окно редактора **Editing WebModule Actions**, в котором можно создавать и удалять действия-сервисы. Нажмем кнопку **Add New**. В окне редактора появится новый сервис — WebActionItem1, который представляет собой объект класса TWebActionItem. Свойства и события этого объекта можно редактировать в инспекторе объектов. Назначим свойству Default объекта WebActionItem1 значение true. Благодаря этому новый сервис станет сервисом по умолчанию (т. е. он будет вызываться при вызове приложения cgidemo без указания сервиса -/cgi-bin/cgidemo). Свойству PathInfo назначим значение "/". Свойство PathInfo идентифицирует новый сервис. С указанным значением "/" имя сервиса будет /cgi-bin/cgidemo/. В поле Producer укажим компонент PageProducer1.

С созданием нового сервиса мы закончили. Обратимся теперь к объекту PageProducer1. В свойстве HTMLFile этого объекта мы должны указать файл страницы-шаблона. Создадим шаблон HTML-страницы, содержащий текст, приведенный в листинге 4.9.

Листинг 4.9. Текст страницы-шаблона

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>CGI-страница</TITLE>
  </HEAD>
  <BODY>
     <H3>Эта страница создана CGI-приложением на основе статического↵
     шаблона</H3>
     <P ALIGN="Center"><FONT SIZE=3>
     <TABLE BORDER=2 BGCOLOR=#e0e0e0>
        <TR>
           <TD>
              <P>Текущее время: <#CT></P>
              <P><FONT COLOR=#000080><B><#HelloStr></B></FONT></P>
           </TD>
        </TR>
     </TABLE>
     </FONT></P>
```

```
  </BODY>
</HTML>
```

Сохраним эту страницу на диске под именем cgihello.html. Наверное, обратили на себя внимание тэги вида <# ...>. Это и есть тэги HTML-шаблона (в терминологии Borland эти тэги иногда еще называются "прозрачными", это означает, что они не имеют смысла для Web-браузера, а предназначены исключительно для шаблонов страниц). Встретив такой тэг в шаблоне страницы, компонент `PageProducer` заменит его соответствующим значением. Откуда компонент берет нужное значение?

Общая структура тэга шаблона имеет вид:

```
<#имя_тэга [параметр1=значение] [параметр2=значение] ...>
```

где имена тэга и параметров являются произвольными (квадратные скобки означают, что указывать параметры тэга необязательно). При получении запроса на генерацию HTML-страницы компонент-генератор сканирует шаблон, и, найдя в нем тэг, начинающийся с <#, вызывает обработчик своего единственного события `OnHTMLTag`. Обработчику передается имя тэга, список параметров тэга, если они есть, и ссылка на строку `ReplaceText`, в которой обработчик должен вернуть HTML-текст, заменяющий данный тэг. Таким образом, функция предоставления значений взамен тэгов шаблона целиком возложена на обработчик `OnHTMLTag`.

Итак, присвоим свойству `HTMLFile` значение cgihello.html и напишем обработчик события `OnHTMLTag` (листинг 4.10)

Листинг 4.10. Обработчик события `OnHTMLTag`

```
void __fastcall TWebModule1::PageProducer1HTMLTag(TObject *Sender,
      TTag Tag, const AnsiString TagString, TStrings *TagParams,
      AnsiString &ReplaceText)
{
  unsigned short hr, min, dummy;
  TDateTime DT = GetTime();
  DecodeTime(DT, hr, min, dummy, dummy);
  if (TagString == "CT")
  {
    ReplaceText = IntToStr(hr)+":"+IntToStr(min);
  }
  else
  if (TagString == "HelloStr")
  {
    if (hr > 12)
```

```
    {
      if (hr < 16) ReplaceText = "Добрый день!";
      else ReplaceText = "Добрый вечер!";
    }
    else
    {
      if (hr > 4) ReplaceText = "Доброе утро!";
      else ReplaceText = "Доброй ночи!";
    }
  }
}
```

Из этого листинга должно быть очевидно, что тэг <#CT> заменяется текущим значением времени, а тэг <#HelloStr> — пожеланием, соответствующим времени суток. На рис. 4.4 показана страница, созданная приложением cgidemo.

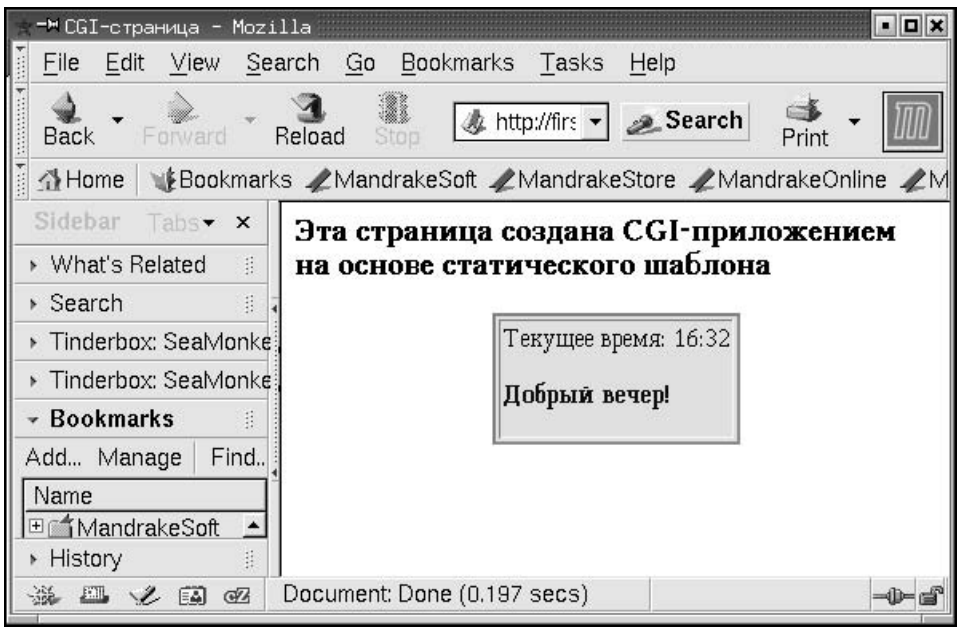


Рис 4.4. Динамическая CGI-страница

В нашем приложении запрос на генерацию страницы передается компоненту-генератору неявно. Вообще же генерация страницы выполняется при обращении к свойству Content компонента-генератора. Например, присваивание:

```
HTMLText := PageProducer.Content;
```

заставляет компонент-генератор сканировать назначенный ему шаблон страницы, заменять в нем тэги шаблона при помощи события `OnHTMLTag` и вернуть полученную HTML-страницу в качестве результата.

Основные принципы технологии WebSnap

Технологию WebSnap можно рассматривать как дальнейшее развитие технологии WebBroker, появившейся еще в среде Delphi 3.

Среди основных возможностей технологии WebSnap нужно отметить следующие:

- поддержка сценариев на стороне сервера (server-side scripting);
- компоненты-Web-модули, являющиеся контейнерами для других компонентов WebSnap и служащие основой для генерации динамических страниц;
- компоненты-диспетчеры, осуществляющие маршрутизацию вызовов и управляющие процессами генерации динамических страниц;
- компоненты-адаптеры, расширяющие возможности сценариев на стороне сервера;
- мастера (wizards) Web-приложений, существенно упрощающие создание заготовки WebSnap-проектов.

Для того чтобы понять смысл сценариев, выполняемых на стороне сервера, следует понять общую "идеологию" уже упоминавшихся компонентов-генераторов контента.

Одна из задач компонентов-генераторов заключается в том, чтобы, по возможности, вывести описание содержимого динамических HTML-страниц, предоставляемых CGI-приложением, за пределы самого приложения. Поскольку статическая часть страницы содержится в шаблоне, хранящемся в отдельном файле, администратор сервера получает возможность модифицировать динамические страницы независимо от CGI-приложения.

Поддержка сценариев на стороне сервера развивает эту тенденцию. Начиная с Kylix 2, у компонентов-генераторов появилось свойство `ScriptEngine`. Поэтому, кроме специальных тэгов, в шаблоны можно динамически включать сценарии, написанные на языке JavaScript. В тексте шаблона блоки сценариев выделяются элементами `<%` и `%>`. В качестве примера приведем сценарий, выводящий первые 10 чисел Фибоначчи (листинг 4.11).

Листинг 4.11. Пример сценария на стороне сервера

```
<%
  fibp = 0;
  fib = 1;
  Response.Write("No 1 : 1<BR>");
  for (i = 2; i <= 10; i++)
  {
    fib = fib + fibp;
    fibp = fib – fibp;
    Response.Write("No "+i+" : "+fib+"<BR>");
  }
%>
```

Для обработки сценариев в Kylix используется движок JavaScript, который реализован в виде отдельной динамической библиотеки. Если наше приложение использует сценарии на стороне сервера, нам придется распространять его вместе с этой библиотекой. Как и любой специальный элемент шаблона, сценарий на стороне сервера будет заменен в результирующей HTML-странице своим значением. В данном случае это будет последовательность чисел Фибоначчи (листинг 4.12).

Листинг 4.12. Результат выполнения сценария из листинга 4.7

```
No 1 : 1<BR>No 2 : 1<BR>No 3 : 2<BR>No 4 : 3<BR>No 5 : 5<BR>No 6 :
8<BR>No 7 : 13<BR>No 8 : 21<BR>No 9 : 34<BR>No 10 : 55<BR>
```

Можно утверждать, что технология WebSnap гораздо теснее связана с использованием шаблонов страниц, нежели технология WebBroker. По этой причине, кроме базовых модулей-контейнеров WebSnap-компонентов (`WebAppDataModule`), в рамках технологии WebSnap используются специальные модули страниц (`WebAppPageModule`). Модули страниц по умолчанию включают в себя компоненты-генераторы контента, связанные с шаблонами страниц. Обычно для каждой динамической страницы приложения создается отдельный модуль.

Компоненты-диспетчеры выполняют маршрутизацию вызовов, направляя запрос соответствующему модулю. Кроме компонента `WebDispatcher`, использующегося также и технологией WebBroker, WebSanp вводит два новых компонента: `PageDispatcher` и `AdapterDispatcher`. Компонент `PageDispatcher` выполняет маршрутизацию вызовов, связанных с модулями страниц, а компонент `AdapterDispatcher` — маршрутизацию вызовов, связанных с командами компонентов-адаптеров *(подробную информацию о командах адаптеров см. в гл. 8)*.

Выше говорилось, что сценарии на стороне сервера повышают гибкость шаблонов динамических страниц. Введение подобных сценариев не принесло бы существенной пользы, если бы они не могли взаимодействовать с серверным приложением, использующим шаблон. Значительная часть тех-

нологии WebSnap сосредоточена на том, чтобы интегрировать сценарии в шаблонах страниц с объектной моделью серверных приложений. В этом смысле технология WebSnap чем-то напоминает поддержку скриптов современными браузерами. Сценарии (скрипты), вставленные в HTML-страницы, выполняются браузерами, при этом сценарии могут использовать ряд объектов браузера. Технология WebSnap реализует тот же принцип на стороне сервера.

Как же предоставить сценариям на стороне сервера доступ к объектам серверного приложения? Для этой цели технология WebSnap использует компоненты-адаптеры. Они играют роль посредников между объектами приложения и сценариями в шаблонах страниц. Например, компонент `ApplicationAdapter` предоставляет сценариям объект `Application`, свойства которого содержат основные данные о серверном приложении. Если серверное приложение использует компонент `ApplicationAdapter`, сценарии в шаблонах страниц могут обращаться к свойствам объекта `Application`. В этом случае строка в странице-шаблоне

```
<h1><%= Application.Title %></h1>
```

будет заменена в готовой странице строкой

```
<h1>Название_приложения</h1>
```

где `название_приложения` — строка, содержащаяся в свойстве `Title` объекта `Application`. Кроме объекта `Application` сценариям доступен еще целый ряд объектов, которые часто позволяют перенести функции обработки запросов в шаблоны динамических страниц и вынести их таким образом за пределы приложения.

Однако функции компонентов-адаптеров не ограничиваются простым предоставлением страницам-шаблонам доступа к объектам. Компоненты адаптеры не только содержат заранее определенные наборы свойств, но и предоставляют программисту возможность определять новые свойства, которые также становятся доступны сценариям на стороне сервера.

В приложениях WebSnap можно использовать несколько типов компонентов-генераторов контента, самым простым из которых является `PageProducer`. Другие компоненты-генераторы, используемые в WebSnap, позволяют работать с наборами данных и документами в форматах XML/XSL. Следует также отметить, что в том, что касается компонентов-генераторов контента, между технологиями WebSnap и WebBroker нет четкой грани. Обе технологии могут использовать одни и те же компоненты-генераторы.

Принципиальная схема взаимодействия компонентов WebSnap-приложения, основанного на модуле страницы, показана на рис. 4.5.

Приведем конкретный пример использования WebSnap в серверном приложении. В предыдущем разделе мы написали CGI-программу, выводящую текущее время и доброе пожелание, зависящее от времени суток. Напишем теперь такую же программу средствами WebSnap.

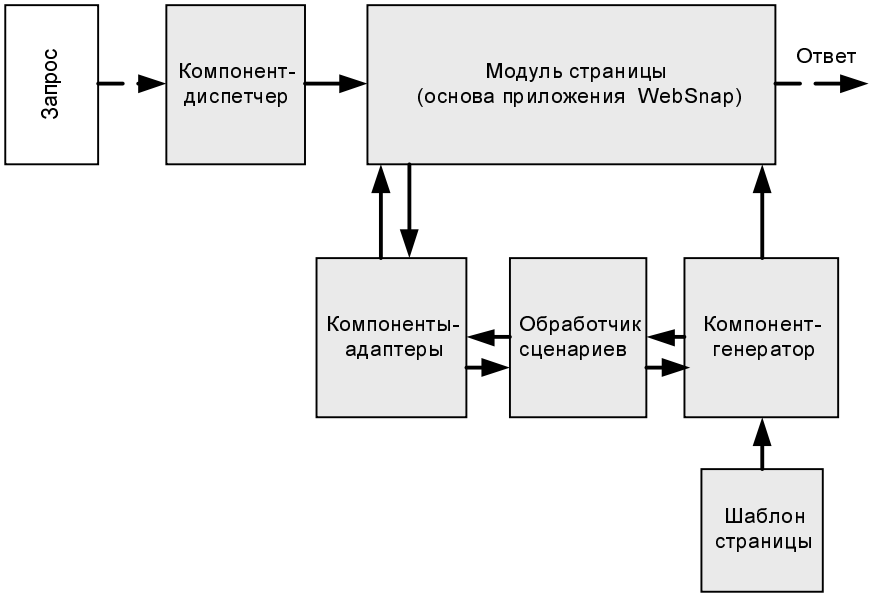


Рис 4.5. Схема взаимодействия компонентов приложения WebSnap (стрелки указывают направление передачи данных)

Прежде всего необходимо создать заготовку WebSnap-приложения. На странице **WebSnap** диалогового окна **New Items** выберем пункт **WebSnap Application**. В открывшемся диалоговом окне можно указать следующие параметры нового проекта:

- **Server Type** — тип приложения (выбираем **CGI stand-alone executable**);
- **Application Module Components** — тип компонентов-модулей (выбираем **Page Module**);
- **Page Name** — название страницы (вводим **GetTimePage**);
- в диалоговом окне **Application Module Page Options** (кнопка **Page Options**) выбираем тип компонента-генератора контента — `PageProducer`.

Остальные опции оставляем без изменений.

После этого будет создан модуль страницы, являющийся главной формой нашего приложения (рис. 4.6).

Как видим, в форме уже размещены основные компоненты приложения WebSnap: генератор контента (`PageProducer`), компонент-адаптер (`ApplicationAdapter`) и компоненты-диспетчеры (`PageDispatcher` и `AdapterDispatcher`). Мы видим здесь также не упоминавшийся ранее компонент `WebAppComponents`. Этот компонент содержит ссылки на все WebSnap-компоненты приложения. При добавлении нового WebSnap-компонента его необходимо зарегистрировать в компоненте `WebAppComponents` (обычно это делается автоматически).

Рис. 4.6. Форма модуля страницы WebSnap-приложения

Сохраним новый проект под именем GetTime. Хотите — верьте, хотите — нет, но простейшее приложение WebSnap уже готово. Можно скомпилировать это приложение, разместить его в каталоге cgi-bin и посмотреть, как оно работает (для этого нужно будет в строке адреса браузера набрать **http://localhost/cgi-bin/GetTime**). Естественно, наше приложение не делает пока ничего полезного, это всего лишь заготовка.

Выше отмечалось, что компонент `ApplicationAdapter` позволяет определить новые свойства, доступные в сценариях шаблонов. Щелкнув кнопкой мыши в поле свойства `Data` этого компонента, можно вызвать редактор полей данных адаптера (рис. 4.7).

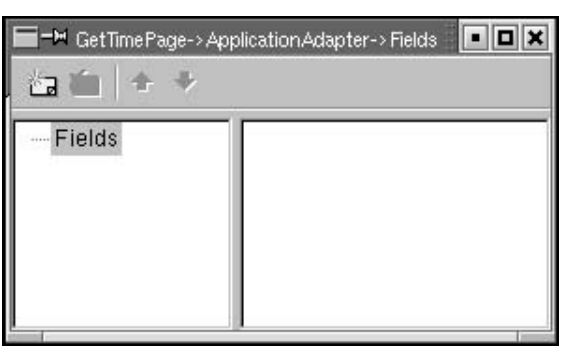


Рис. 4.7. Редактор полей компонента `ApplicationAdapter`

Нажмем кнопку **New Component**. На экране появится окно **Add Web Component** (рис. 4.8), в котором будет предложено выбрать тип создаваемого поля. Выберем тип `AdapterMemoField`.

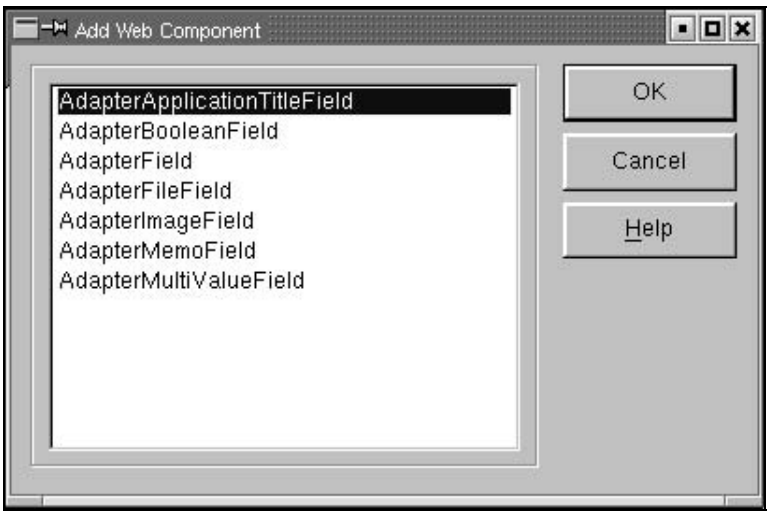


Рис 4.8. Окно **Add Web Component**

Присвоим новому полю имя `Hrs` (часы) и создадим еще одно, такое же, присвоив ему имя `Mins` (минуты). После этого окно редактора полей должно выглядеть, как на рис. 4.9.

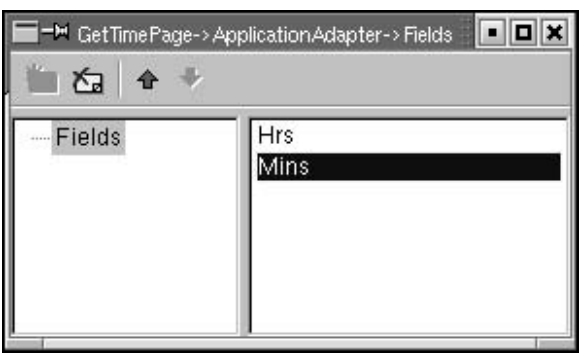


Рис 4.9. Редактор полей с полями `Hrs` и `Mins`

Поля `Hrs` и `Mins` являются объектами, и их свойства можно редактировать в инспекторе объектов. Назначим свойствам `OnGetValue` компонентов `Hrs` и `Mins` обработчики, приведенные в листинге 4.13.

Листинг 4.13. Обработчики `OnGetValue` компонентов `Hrs` и `Mins`

```
void __fastcall TGetTimePage::MinsGetValue(TObject *Sender,
AnsiString &Value)
{
  unsigned short hr, min, dummy;
  TDateTime dt = GetTime();
  DecodeTime(dt, hr, min, dummy, dummy);
  Value = IntToStr(min);
}
//---------------------------------------------------------------------
void __fastcall TGetTimePage::HrsGetValue(TObject *Sender,
AnsiString &Value)
{
  unsigned short hr, min, dummy;
  TDateTime dt = GetTime();
  DecodeTime(dt, hr, min, dummy, dummy);
  Value = IntToStr(hr);
}
//---------------------------------------------------------------------
```

Из листинга 4.13 очевидно, что поле `Hrs` возвращает текущее значение часов, а поле `Mins` — значение минут.

Само приложение GetTime готово, осталось отредактировать шаблон страницы. Заготовка шаблона была создана в процессе создания заготовки проекта. Имя страницы должно совпадать с именем программного модуля (не надо путать его с Web-модулем страницы), содержащего код обработчиков. Стандартная заготовка шаблона страницы содержит много такого, что нам сейчас не нужно, поэтому заменим ее текст на следующий (листинг 4.14).

Листинг 4.14. Страница-шаблон для приложения GetTime

```
<html>
<head>
<title>
<%= Page.Title %>
</title>
</head>
<body>
<h1>Get Time</h1>
  <P><U><A href="<% = Page.HREF%>">Обновить</A></U></P>
```

```
  <P>Текущее время <%=Application.Hrs.Value%>:
    <% if (Application.Mins.Value < 10) Response.Write("0");%>
    <%=Application.Mins.Value%></P>
    <% if (Application.Hrs.Value > 12)
    {
      if (Application.Hrs.Value < 16) ReplaceText = "Добрый день!";
      else ReplaceText = "Добрый вечер!";
    }
    else
    {
      if (Application.Hrs.Value > 4) ReplaceText = "Доброе утро!";
      else ReplaceText = "Доброй ночи!";
    } %>
    <P><B><% =ReplaceText%></B></P>
    <P>Каталог сервера: <%=Application.QualifyFileName("./")%></P>
</body>
</html>
```

Эта страница иллюстрирует использование в сценариях на стороне сервера как стандартных полей объектов, так и тех полей, которые были введены нами в процессе написания нашего приложения.

Анализируя текст сценария на стороне сервера, важно понять принципиальную разницу между полями объектов WebSnap и динамическими тэгами WebBroker. В станице-шаблоне WebSnap поле объекта не заменяется автоматически своим значением. Включить значение поля объекта в текст конечной HTML-страницы можно, например, при помощи конструкции

```
<% = Method.Field %>
```

Можно также воспользоваться методом `Write` объекта `Response`. Метод `Write` может понадобиться также для включения в текст конечной страницы константного значения из тела сценария.

Работая с собственными полями, следует помнить, что с точки зрения объектной модели сценариев на стороне сервера, переменные `Hrs` и `Mins` являются объектами, и выполнять с ними непосредственно операции присваивания нельзя. Для получения значений `Hrs` и `Mins` необходимо воспользоваться свойством `Value` класса `AdapterField`. Это самое важное различие между предустановленными полями объектов WebSnap и полями, введенными программистом.

Объектная модель WebSnap позволяет обращаться не только к полям, но и к методам объектов. В качестве примера использования метода объекта `Application` мы приводим строку

```
<%=Application.QualifyFileName("./")%>
```

Эта строка выводит в текст конечной страницы полный путь к каталогу, в котором расположено серверное приложение WebSnap. Метод `QualifyFileName` объекта `Application` транслирует переданный ему в качестве параметра относительный путь в абсолютный. При этом текущим каталогом считается каталог приложения (в нашем случае — `cgi-bin`). Конечно, программист, заботящийся о сетевой безопасности, вряд ли захочет предоставлять удаленным пользователям информацию о том, где именно расположено его приложение. Тем не менее метод `QualifyFileName` может быть весьма полезен, например, если наше приложение работает с файловой системой при помощи функций, которым необходим полный путь к файлу.

Для того чтобы проверить работу приложения GetTime, его необходимо скопировать в каталог /cgi-bin/. Кроме этого нам потребуется скопировать в каталог системных библиотек (/usr/lib/) библиотеку libjs.so из каталога kylix3/bin/. Это та самая библиотека, в которой реализован обработчик сценариев.

Набрав после этого в строке браузера **http://localhost/cgi-bin/GetTime**, мы должны увидеть страницу, подобную той, что показана на рис. 4.10.

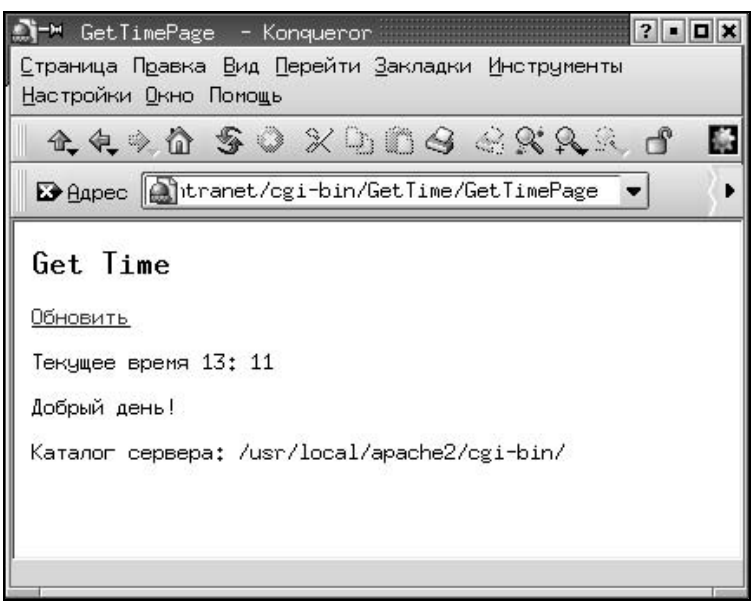


Рис 4.10. Страница, созданная приложением GetTime

Сравнивая листинги 4.10 и 4.13, мы видим, что в WebSnap-приложении многие функции обработчиков событий можно вынести в шаблоны страниц, которые в свою очередь могут модифицироваться отдельно от основного приложения.

В рамках технологии WebBroker мы могли изменять лишь статическое "окружение" вокруг динамических тэгов. Технология WebSnap предоставляет шаблонам доступ к полям и методам объектов приложения, а также позволяет использовать эти объекты в сценариях, написанных на языке JavaScript. Таким образом, степень свободы при работе с шаблонами в рамках технологии WebSnap гораздо выше, чем при использовании технологии WebBroker.

В этом и заключается преимущество технологии WebSnap.

SOAP — технология распределенных объектов для Web

В этом разделе речь пойдет о технологиях, объединенных под общим названием Web Services (Web-службы). Kylix позволяет создавать приложения Web-служб на основе протокола SOAP. Протокол SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) предназначен для организации взаимодействия между распределенными приложениями на основе таких распространенных протоколов, как HTTP и XML. Можно сказать, что протокол SOAP представляет собой дальнейший шаг в развитии средств двустороннего взаимодействия между Web-приложениями, первым из которых был интерфейс CGI. Одно из фундаментальных различий между CGI и SOAP заключается в том, что CGI представляет собой средство обмена данными между HTML-браузером и Web-сервером, в то время, как протокол SOAP позволяет организовать обмен данными между приложениями любых типов и практически не связан с HTML.

В рамках протокола SOAP приложение-сервер предоставляет приложению-клиенту доступ к методам ряда своих объектов. Клиент посылает серверу запрос, содержащий команды на выполнение соответствующих методов, и необходимые методам данные, а в ответ получает данные, возвращенные вызванным методом. Для организации обмена данными при помощи некоторого SOAP-объекта, клиент SOAP должен "знать" интерфейс объекта, экспортируемого сервером. Под интерфейсом в данном контексте понимается совокупность экспортируемых методов объекта, списки параметров методов и типы возвращаемых значений. Поскольку SOAP позволяет организовать взаимодействие между компонентами распределенной системы, написанными на разных языках программирования и выполняющимися на разных платформах, необходим платформо-независимый язык описания интерфейсов объектов. Для описания экспортируемых интерфейсов в протоколе SOAP используется спецификация WSDL. Аббревиатура WSDL расшифровывается, как Web Services Description Language — язык описания Web-служб (в терминологии SOAP экспортируемые объекты называются Web-службами). Обычно SOAP-сервер позволяет удаленным клиентам загружать описания экспортируемых объектов на языке WSDL посредством

протокола HTTP, благодаря чему SOAP-клиенты могут создавать интерфейсы для взаимодействия с сервером динамически, во время выполнения.

Поскольку SOAP основан на протоколе HTTP, приложения-серверы SOAP очень удобно реализовать в форме приложения для Web-сервера.

Как и в случае с CGI, приложения-серверы SOAP могут быть написаны на стандартном C++ (или стандартном Delphi Language), без использования специальных элементов среды разработки, но как и в случае с CGI, использование специальных элементов существенно упрощает процесс разработки.

Для того, чтобы создать приложение-сервер SOAP, следует выбрать пункт **SOAP Server Application** на вкладке **WebServices** диалогового окна **New Items**. Как и другие приложения для Web-сервера, сервер SOAP можно разрабатывать в одном из трех вариантов:

- в виде независимого CGI-приложения;
- в виде разделяемого модуля для сервера Apache;
- в виде специального приложения для встроенного отладчика.

После выбора одного из вариантов сервера Kylix автоматически создает его главный модуль. На следующий за этим вопрос о том, хотим ли мы создать интерфейс для модуля SOAP, отвечаем отрицательно. Форма главного модуля сервера SOAP (рис. 4.11) содержит три основных компонента — `HTTPSoapDispatcher`, `HTTPSoapPascalInvoker` (для Delphi Language) или `HTTPSoapCppInvoker` (для языка C++) и `WSDLHTMLPublish`. Компонент `HTTPSoapDispatcher` выполняет маршрутизацию вызовов SOAP, `HTTPSoapPascalInvoker` отвечает за вызов соответствующих методов объектов Kylix, а `WSDLHTMLPublish` динамически генерирует описание интерфейса в формате WSDL в ответ на запрос клиента.

Сохраним наш проект под именем SoapServ.

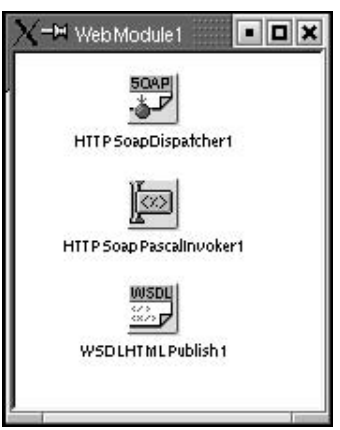


Рис 4.11. Главный модуль приложения-сервера SOAP

С точки зрения приложения-сервера, объекты SOAP представляются обычными объектами языка C++ или Delphi Language. Для того чтобы экспортировать объект по протоколу SOAP, необходимо объявить интерфейс, имеющий то же имя, что и объект, экспортирующий методы объекта.

Для добавления нового экспортируемого объекта в приложение-сервер на той же вкладке **WebServices** выбираем пункт **SAOP Server Data Module**. При этом выводится диалоговое окно, в котором нам предлагается ввести имя нового интерфейса. В качестве имени объекта введем `GetHello`. После этого в проект будет добавлена новая форма (наследник класса `TSoapDataModule`) и новый файл исходного текста, в котором уже объявлен класс `TGetHello` с уникальным идентификатором и соответствующий ему интерфейс `IGetHello`. Кроме того, в новом модуле вызываются функции, регистрирующие объект и интерфейс в реестре экспортируемых объектов приложения. К этому реестру обращаются компоненты `HTTPSoapDispatcher` и `WSDLHTMLPublish`.

Дальнейшее программирование нашего приложения-сервера сводится к добавлению новых методов в экспортируемый объект (разумеется, более сложные приложения могут экспортировать несколько объектов, заданных программистами). Экспортируемые методы должны располагаться в разделе `public` соответствующего класса. Кроме того, объявление экспортируемого метода следует добавить в описание интерфейса. Экспортируемые методы должны использовать формат вызова `stdcall`. Следует учитывать, что приложения-серверы SOAP не должны полагаться на сохранение информации между отдельными вызовами. Также как и приложения CGI, серверы SOAP вызываются специально для каждого запроса со стороны клиента. В перерывах между вызовами связь между клиентом и сервером не поддерживается и состояние сервера не сохраняется.

Добавим в раздел `public` класса `TGetHello` метод `GetHello` с единственным параметром типа `String` и возвращающий значение того же типа. Параметром метода `GetHello` является имя удаленного клиента. Если удаленный клиент передает методу `GetHello` строку `"ClientName"`, метод возвращает строку `"Hello ClientName!"`.

В листингах 4.15 и 4.16 приводятся реализации класса `TGetHello` и интерфейса `IGetHello` на языке C++, а в листинге 4.17 — на Delphi Language (файлу модуля присвоено имя GetHelloU).

Листинг 4.15. Вариант реализации класса `TGetHello` и интерфейса `IGetHello` на языке C++ (заголовочный файл GetHelloU.h, фрагмент)

```
//************************************************************************ //
// SOAP Data Module: Interface GetHello
//************************************************************************ //


#ifndef  GetHelloUH
```

```
#define GetHelloUH

#include <InvokeRegistry.hpp>
#include <Midas.hpp>
#include <SOAPMidas.hpp>
#include <SOAPDm.hpp>

//************************************************************************ //
// Экспортируемый SOAP-интерфейс
//************************************************************************ //
__interface IGetHello : public IAppServerSOAP
{
  /* Этот метод мы добавляем в класс TGetHello */
  String __stdcall GetHello(String Name);
};
typedef DelphiInterface<IGetHello> _di_IGetHello;

//************************************************************************ //
// Класс, реализующий экспортируемые методы
//************************************************************************ //
class TGetHello : public TSoapDataModule, public IGetHello, public
IAppServerSOAP
{
__published:
private:
public:
  __fastcall TGetHello(TComponent* Owner);
  String __stdcall GetHello(String Name);

  /* IAppServerSOAP */
  ...

 /* IUnknown Methods */
  ...
};

#endif
```

Листинг 4.16. Вариант реализации класса `TGetHello` и интерфейса `IGetHello` на языке C++ (файл GetHelloU.cpp, фрагмент)

```
...
//---------------------------------------------------------------------- //
// Реализация методов TGetHello, объявленных в интерфейсе IGetHello
//---------------------------------------------------------------------- //
__fastcall TGetHello::TGetHello(TComponent* Owner) :
TSoapDataModule(Owner)
{}

String __stdcall TGetHello::GetHello(String Name)
{
  return "Hello" + Name + "!";
}
...
void static RegTypes()
{
  InvRegistry()->RegisterInterface(__interfaceTypeinfo(IGetHello));
  InvRegistry()->RegisterInvokableClass(__classid(TGetHello),
  GetHelloFactory);
}
```

Листинг 4.17. Вариант реализации класса `TGetHello` и интерфейса `IGetHello` на языке Delphi Language

```
Unit GetHelloU;

interface

uses SysUtils, Classes, InvokeRegistry, Midas, SOAPMidas, SOAPDm;

type
  IGetHello = interface(IAppServer)
    ['{627DBBDC-40C4-D611-9FD8-70164708DC97}']
    (* Этот метод мы добавляем в класс TGetHello *)
    function GetHello(Name : String) : String; stdcall;
  end;
```

```
  TGetHello = class(TSoapDataModule, IGetHello, IAppServer)
  private

  public
    function GetHello(Name : String) : String; stdcall;
  end;

implementation

{$R *.xfm}

procedure TGetLogCreateInstance(out obj: TObject);
begin
 obj := TGetHello.Create(nil);
end;

function TGetHello.GetHello(Name : String) : String;
begin
  Result := 'Hello ' + Name + '!';
end;

initialization
   InvRegistry.RegisterInvokableClass(TGetHello, TGetLogCreateInstance);
   InvRegistry.RegisterInterface(TypeInfo(IGetHello));
end.
```

В приведенных листингах все, кроме объявления и описания метода `GetHello`, сгенерировано средой разработки автоматически, и нам нет особой необходимости вникать в эти детали.

Фактически простейшее приложение SOAP уже готово, но для того, чтобы проверить его работу, нам придется выполнить некоторые предварительные действия. Прежде всего, нам придется изменить настройки сервера Apache. Как уже отмечалось, SOAP-сервер предоставляет удаленным клиентам информацию об экспортируемых объектах (службах). Стандартное SOAP-приложение, созданное в среде Borland Kylix, использует для обработки таких запросов порт 8080, который, скорее всего, не прослушивается нашим Web-сервером. Для того чтобы сервер Apache начал прослушивать этот порт, необходимо отредактировать файл httpd.conf (кто устанавливал сервер

Apache самостоятельно, то наверняка уже имел дело с этим файлом). Добавим в файл httpd.conf строку

```
Listen 127.0.0.1:8080
```

где вместо локального адреса `127.0.0.1` можно указать IP-адрес нашего узла (если такой имеется), и перезапустим сервер.

Кроме этого, нам придется установить в каталоге /usr/lib/ ряд библиотек, в которых нуждается SOAP-сервер. В список необходимых библиотек входят следующие:

- libxercesxmldom.so.1;
- libxercesxsldom.so.1;
- libxerces-c1_6_0.so;
- libxalan-c1_3.so;
- libicuuc.so.20;
- libicudt20l.so.

Все эти библиотеки можно найти в каталоге kylix3/bin/.

Теперь можно скопировать CGI-модуль `SOAPServ` в каталог /cgi-bin/ и вызвать его в окне браузера. Как видим (рис. 4.12), наш SOAP-сервер выводит довольно много информации в формате HTML. Эта информация предназначена не для пользователя, а для программиста, желающего написать SOAP-клиент для нашего сервера.

На странице **Service Info Pages** приводится список интерфейсов (сервисов), предоставляемых сервером SOAP. Кроме созданного нами интерфейса `IGetHello`, сервер экспортирует ряд дополнительных интерфейсов (`IAppServer`, `IAppServerSOAP`). Назначение этих интерфейсов будет объяснено в главе 9. Для того, чтобы получить более подробные сведения о каком-либо интерфейсе, следует щелкнуть на ссылке, указывающей на интерфейс. Например, перейдя по ссылке **IGetHello**, мы получим подробный перечень методов интерфейса `IGetHello` (рис. 4.13).

Последним элементом списка методов `IGetHelo` является введенный нами метод `GetHello`. Остальные методы, представленные в списке, унаследованы интерфейсом `IGetHello` от его предков.

Теперь напишем клиент, взаимодействующий с нашим SOAP-сервером. Для того, чтобы разрабатываемый клиент мог взаимодействовать с сервером, нам нужно объявление интерфейса, предоставляемого сервером. Поскольку мы сами писали сервер, объявление этого интерфейса у нас уже есть. Мы можем скопировать объявление интерфейса `IGetHello` из модуля `GetHelloU` в новый модуль и использовать этот модуль при написании приложения-клиента. В листинге 4.18 приводится исходный текст модуля `GetHelloIntf`, содержащего объявление интерфейса `IGetHello` (модуль написан на Delphi Language).

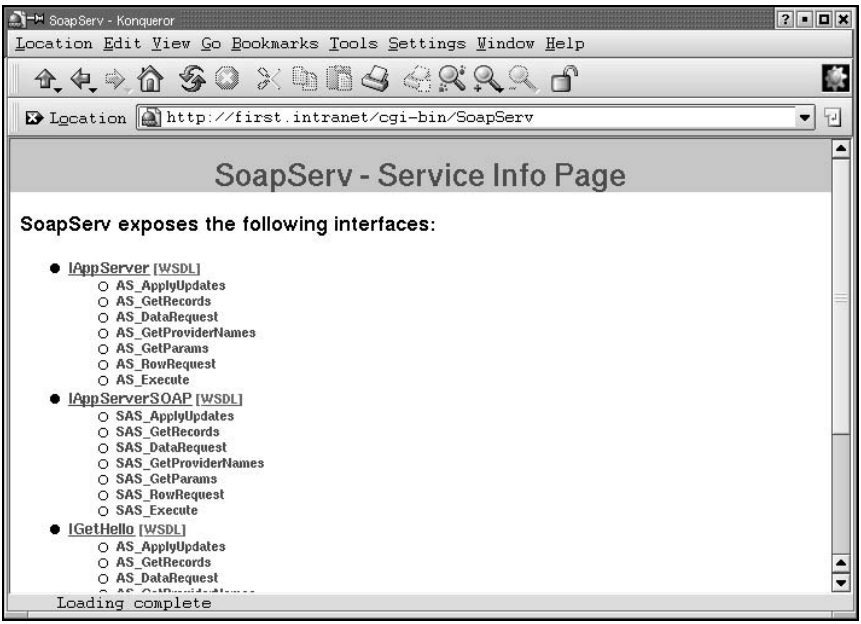


Рис 4.12. Страница, содержащая информацию об интерфейсах, экспортируемых сервером SOAP

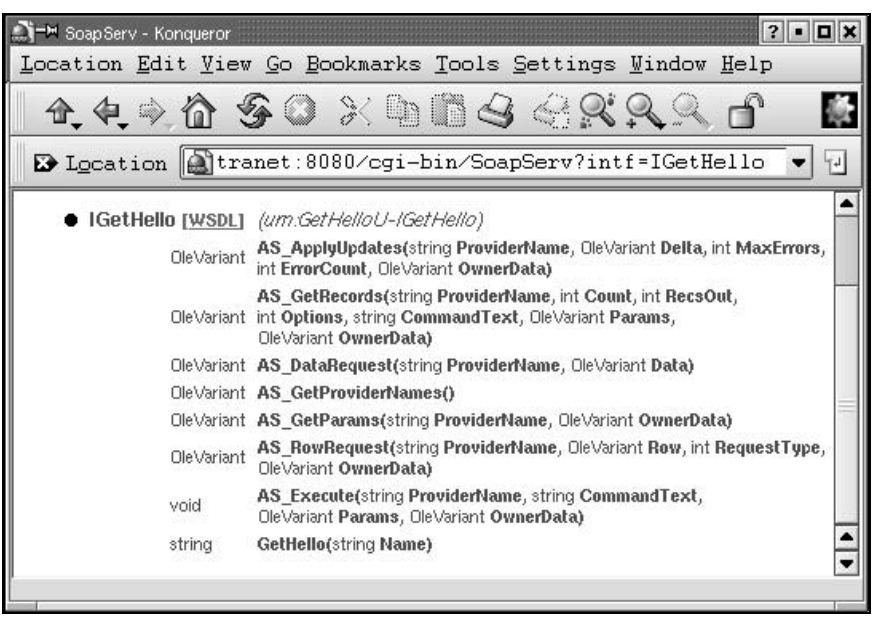


Рис 4.13. Список методов интерфейса IGetHello

Листинг 4.18. Модуль `GetHelloIntf`, полученный копированием описания интерфейса

```
unit GetHelloIntf
interface

uses SysUtils, Classes, InvokeRegistry, Midas, SOAPMidas, SOAPDm;

type
  IGetHello = interface(IAppServer)
    ['{627DBBDC-40C4-D611-9FD8-70164708DC97}']
    function GetHello(Name : String) : String; stdcall;
  end;

implementation

initialization
  InvRegistry.RegisterInterface(TypeInfo(IGetHello));
end.
```

При всей своей простоте такой подход имеет очевидные недостатки. У нас может не быть доступа к исходным текстам приложения-сервера, либо сервер может быть написан на другом языке программирования.

Выше говорилось, что серверы SOAP предоставляют описания своих интерфейсов на языке WSDL. Работая в Kylix, мы можем использовать WSDL-описание интерфейса для автоматической генерации программных модулей, содержащих объявление интерфейса на соответствующем языке программирования.

Для того, чтобы загрузить WSDL-описание, щелкнем на ссылке **[WSDL]** справа от имени интересующего нас интерфейса (`IGetHello`). В окне браузера ничего нет, но это не означает, что произошла ошибка. Страница WSDL браузером загружена, просто он не может отобразить ее визуально. Сохраним чистую страницу на диске под именем IGetHello.wsdl (можно убедиться, что сохраненный файл занимает около 8 Кбайт).

Теперь можно преступить к написанию приложения-клиента. Приложение-клиент SOAP является обычным приложением Kylix. Создадим заготовку стандартного приложения в среде Delphi Language. Для того, чтобы наше приложение стало SOAP-клиентом, нам необходимо объявление интерфейса `IGetHello`. Для его получения воспользуемся файлом IGetHello.wsdl. На вкладке **WebServices** диалогового окна **New Items** выбираем пункт **WSDL Importer**. При этом будет запущен мастер **WSDL Import Wizard**, на первой странице которого нам будет преложено указать расположение файла с

WSDL-описанием. На следующей странице мастера можно выполнить предварительный просмотр и редактирование создаваемого модуля. После нажатия кнопки **Finish** в проект будет добавлен новый модуль (листинг 4.19), содержащий объявление интерфейса `IGetHello`.

Листинг 4.19. Автоматически сгенерированное объявление интерфейса `IGetHello`

```
unit IGetHelloIntf;

interface

uses InvokeRegistry, Types, XSBuiltIns;

type
  IGetHello = interface(IInvokable)
  ['{A082F427-17D9-ACAB-5FB6-3BDE8853202B}']
  function  AS_ApplyUpdates(const ProviderName: WideString;
  const Delta: Variant; const MaxErrors: Integer; var OwnerData: Variant;
  return: Variant): Integer; stdcall;
  function  AS_GetRecords(const ProviderName: WideString;
  const Count: Integer; const Options: Integer;
  const CommandText: WideString; var Params: Variant;
  var OwnerData: Variant; return: Variant): Integer; stdcall;
  function  AS_DataRequest(const ProviderName: WideString;
  const Data: Variant): Variant; stdcall;
  function  AS_GetProviderNames: Variant; stdcall;
  function  AS_GetParams(const ProviderName: WideString;
  var OwnerData: Variant): Variant; stdcall;
  function  AS_RowRequest(const ProviderName: WideString;
  const Row: Variant; const RequestType: Integer;
  var OwnerData: Variant): Variant; stdcall;
  procedure AS_Execute(const ProviderName: WideString;
  const CommandText: WideString; var Params: Variant;
  var OwnerData:  Variant); stdcall;
  function  GetHello(const Name: WideString): WideString; stdcall;
  end;

function GetIGetHello(UseWSDL: Boolean=System.False; Addr: string='')
```

```
: IGetHello;

implementation
  uses SOAPHTTPClient;

function GetIGetHello(UseWSDL: Boolean; Addr: string): IGetHello;
const
  defWSDL = '/home/k3/SOAPServ/IGetHello.wsdl';
  defURL  = 'http://first.intranet:8080/cgi-bin/SoapServ/soap/IGetHello';
  defSvc  = 'IGetHelloservice';
  defPrt  = 'IGetHelloPort';
var
  RIO: THTTPRIO;
begin
  Result := nil;
  if (Addr = '') then
  begin
    if UseWSDL then
      Addr := defWSDL
    else
      Addr := defURL;
  end;
  RIO := THTTPRIO.Create(nil);
  try
     if UseWSDL then
    begin
      RIO.WSDLLocation := Addr;
      RIO.Service := defSvc;
      RIO.Port := defPrt;
    end else
      RIO.URL := Addr;
    Result := (RIO as IGetHello);
  finally
    if Result = nil then
      RIO.Free;
  end;
end;
```

```
initialization
  InvRegistry.RegisterInterface(TypeInfo(IGetHello),
  'urn:GetHelloU-IGetHello', 'utf-8');
end.
```

В случае генерации объявления интерфейса для языка C++ интерфейс получит имя `_di_IGetHello`.

Вся работа по взаимодействию с SOAP-сервером выполняется одним-единственным компонентом — `HTTPRIO` (страница **WebServices** палитры компонентов).

Компонент `HTTPRIO` не только выполняет SOAP-запрос, используя протокол HTTP, но и позволяет обращаться к методам интерфейса SOAP-сервера так же, как и к методам обычных объектов C++ или Delphi Language. Для того, чтобы компонент `HTTPRIO` мог выполнить свои функции по работе с интерфейсом, ему необходимо указать расположение WSDL-описания интерфейса (свойство `WSDLLocation`). Из этого описания компонент `HTTPRIO` извлечет не только данные о методах, предоставляемых интерфейсом, но также и информацию о месте нахождения SOAP-сервера в сети и строке запроса, необходимой для получения доступа к интерфейсу. Заметим, что WSDL-описание не обязательно должно храниться в виде файла на локальном диске. В качестве места расположения WSDL-описания интерфейса можно указать и ссылку на динамическую страницу, генерируемую SOAP-сервером, причем, делать это можно и во время выполнения программы. Более сложные методы работы с компонентом `HTTPRIO` будут рассмотрены в главе 9, здесь же мы приведем простейший пример.

Итак, у нас есть описание интерфейса `IGetHello`, и мы хотим, чтобы наш клиент посылал запросы серверу SOAPServ, используя метод `GetHello` этого интерфейса. В листинге 4.20 приводится текст функции `GetServerResponse`, которая использует объект `HTTPRIO1`, настроенный на работу с SOAP-сервером, как было указано выше.

Листинг 4.20. Функция `GetServerResponse` (вариант Delphi Language)

```
function GetServerResponse(const Name : String) : String;
var
  IGH : IGetHello;
begin
  IGH := HTTPRIO1 as IGetHello;
  if Assigned(IGH) then Result := IGH.GetHello(Name)
  else Result := '';
  IGH := nil;
end;
```

Как видно из этого примера, компонент HTTPRIO является полиморфным, предоставляя программисту тот SOAP-интерфейс, с которым этот компонент связан в данный момент. В функции, написанной на Delphi Language, мы использовали оператор as для преобразования типов объектов. Вариант функции GetServerResponse, написанной на языке C++, приводится в листинге 4.21.

Листинг 4.21. Функция GetServerResponse (вариант C++)

```
String GetServerResponse(const String Name)
{
  _di_IGetHello IGH;
  String result;
  HTTPRIO1->QueryInterface(IGH);
  if (IGH) result = IGH->GetHello(Name);
  else result = '';
  IGH = NULL;
  return result;
}
```

Отладка Web-приложений в Kylix 3

В предыдущих разделах этой главы мы довольно подробно рассмотрели программирование различных типов приложений для Web-сервера, однако до сих пор не затронули один очень важный вопрос — отладку серверных приложений. Приложения для сервера могут выполняться нормально, только если они запущены сервером, поэтому стандартные методы отладки для них не применимы. Впрочем, путем некоторых ухищрений, можно организовать отладку приложения для сервера так же, как мы отлаживаем разделяемые библиотеки, т. е. связать проект с исполняющимся приложением.

На этом пути возникают две сложности. Первая сложность связана с тем, что для отладки приложения, выполняющегося в среде сервера, как правило, следует иметь права супер-пользователя (root), а, значит, данный метод отладки можно использовать, только если Kylix установлен в режиме root. Вторая сложность связана с тем, что приложения, вызываемые сервером, выполняются в течение довольно короткого промежутка времени и после этого завершаются, так что скорее всего мы не успеем связать проект с выполняющимся процессом. Для решения этой проблемы можно изменить код приложения, вставив в него цикл, выход из которого зависит от значения некоторой переменной. В самом проекте переменной присваивается значение, делающее цикл бесконечным.

При запуске такого приложения Web-сервером оно входит в бесконечный цикл, и у нас будет достаточно времени для того, чтобы связать проект с процессом. После этого мы с помощью отладчика сможем изменить значение переменной, контролирующей цикл, и выполнение программы продолжится.

Рассмотрим все вышесказанное на конкретном примере, выполнив отладку простого CGI-приложения, представленного в листинге 4.2. Прежде всего, внесем необходимые изменения в исходный текст приложения (листинг 4.22).

Листинг 4.22. Изменения, внесенные в приложение SimpleCGI

```
int main(int argc, char* argv[])
{
  char * Method, * Val;
  Method = getenv("REQUEST_METHOD");
  if (Method == NULL)
  {
    cout << "Ошибка, вы пытаетесь запустить CGI-приложение\n";
    return 1;
  }
  cout << "Content-Type: text/html\n\n";
  /* Код, внесенный для упрощения отладки */
  #ifdef _DEBUG
  bool loop = true;
  while(loop);
  #endif
  cout << "<HTML><HEAD><META HTTP-EQUIV=\"Content-Type\"↳
  CONTENT=\"text/html; charset=koi8-r\">";
  ...
```

Теперь, после вызова приложения /cgi-bin/SimpleCGI, мы сможем связать его с проектом (рис. 4.14).

После связывания процесса с отладчиком мы можем изменить значение переменной `loop` с помощью команды **Evaluate | Modify**. Программа выйдет из цикла, и мы сможем выполнять дальнейшую отладку приложения обычным образом.

Указанный выше метод отладки нельзя назвать очень удобным. Kylix 3 предоставляет в распоряжение программистов новое средство — встроенный отладчик Web-приложений — Web App Debugger.

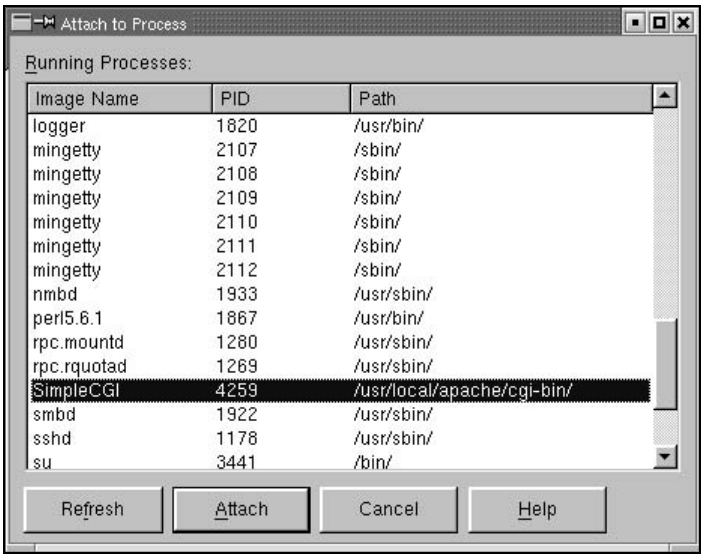


Рис 4.14. Окно **Attach to Process** для CGI-приложения

Отладчик фактически представляет собой связанный с IDE Web-сервер, по умолчанию слушающий порт 8081 (если в нашей системе указанный порт используется другим интернет-сервисом, мы можем изменить это значение в диалоговом окне **Server/Options** приложения Web App Debugger). С помощью отладчика Web App Debuger Web-приложения можно отлаживать точно так же, как и обычные программы.

Web App Debuger взаимодействует с отлаживаемыми приложениями посредством специального интерфейса, поэтому непосредственная отладка независимых CGI-приложений и разделяемых модулей Web-сервера Apache с его помощью невозможна. Для того, чтобы Web-приложение можно было отлаживать с помощью отладчика Web App Debuger, в процессе создания проекта приложения необходимо указать тип приложения **Web App Debugger Executable**. При этом мы должны будем назначить новому проекту имя класса, которое будет использоваться отладчиком для идентификации проекта.

Отличительной особенностью приложений для Web-отладчика является наличие в проекте компонента-наследника `TForm`. Этот компонент необходим для того, чтобы отлаживаемое приложение можно было запускать в среде разработки как обычную программу. В остальном программирование приложений для отладчика ни чем не отличается от программирования приложений других классов.

Когда Web-проект готов к работе, запустим его, как обычное приложение Kylix. В окне редактора кода Web-проекта установим точки останова там, где мы считаем это необходимым, затем запустим Web-отладчик командой

меню **Tools | Web App Debugger**. В открывшемся окне нажмем кнопку **Start** и щелкнем на ссылке **Default URL**. При этом запускается браузер, в котором открывается страница (рис. 4.15), содержащая список всех зарегистрированных отладчиком приложений. Выберем в этом списке наш проект. После нажатия кнопки **Go**, расположенной на странице справа от списка приложений, можно выполнять интерактивную отладку нашего проекта.

Рис 4.15. Страница, содержащая список зарегистрированных серверных приложений

Отладчик Web App Debugger позволяет не только выполнять различные операции (приостанавливать выполнение программы, просматривать и модифицировать значения переменных), но и получить данные о числе обработанных приложением запросов и времени, затраченном на их обработку (вкладка **Statistics**, рис. 4.16), а также просматривать log-файл обращений к серверу (вкладка **Log**).

После окончания отладки нашего проекта, мы, скорее всего, захотим преобразовать его в CGI-приложение или модуль Apache. Для этого можно создать заготовку нового приложения соответствующего типа. С помощью менеджера проектов удалить из него созданные по умолчанию файлы Web-модулей и скопировать файлы Web-модулей приложения, созданного для отладчика, в новый проект.

Но можно пойти и другим путем, а именно, поместить отлаженный Web-модуль в репозиторий объектов и затем вставить его в проект нового приложения. Щелкаем правой кнопкой мыши в окне **WebModule1** (не `Form1`!) и

в контекстном меню выбираем пункт **Add to Repository...** В открывшемся окне вводим в поле **Title** значение **MyApp**, а в поле **Page** выбираем значение **Data Modules**. Можно также задать пиктограмму нового шаблона, описание и имя автора. Нажимаем кнопку **OK**. Теперь создаем новый проект нужного нам типа (CGI-приложение или модуль Apache) и удаляем из него файлы Web-модуля. Затем снова открываем окно **New Items** и на странице **Data Modules** выбираем компонент **MyApp**. После этого отлаженный модуль автоматически переносится в новый проект.

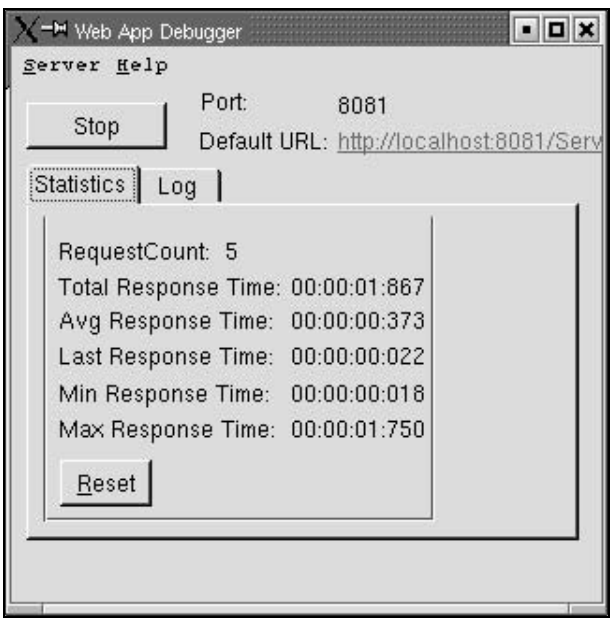


Рис 4.16. Окно приложения Web App Debugger с открытой вкладкой **Statistics**

Глава 5

Работа с компонентами Internet Direct

Эта глава посвящена подробному описанию компонентов Internet Direct, с которыми мы уже познакомились в предыдущей главе. Напомним, что компоненты Internet Direct (Indy) предназначены для разработки независимых интернет-приложений и делятся на четыре категории:

1. Компоненты-клиенты.
2. Компоненты-серверы.
3. Компоненты-перехватчики.
4. Вспомогательные компоненты.

Поскольку в каждой новой версии своих средств разработки компания Borland использует новые версии пакета Internet Direct, набор компонентов Indy для Kylix может отличаться от набора компонентов средств разработки для ОС Windows, с которыми мы работаем в данный момент. Этот факт следует учитывать при рассмотрении возможности переноса Indy-приложений с одной платформы на другую.

Компоненты-клиенты предоставляют как реализацию базовых клиентов протоколов TCP (Transmission Control Protocol, протокол управления передачей) и UDP (User Datagram Protocol, протокол дейтаграмм пользователя), так и различных распространенных интернет-протоколов. Компоненты, реализующие клиенты для протоколов, основанных на TCP, являются потомками базового компонента-клиента TCP, а компоненты-клиенты UDP-протоколов являются потомками базового компонента-клиента UDP.

В табл. 5.1 приводится список компонентов-клиентов Indy для Kylix 3.

Таблица 5.1. *Компоненты-клиенты Indy*

Компонент	Описание
`IdTCPClient`	Базовый клиент протокола TCP. Является предком всех компонентов-клиентов, основанных на TCP
`IdUDPClient`	Базовый клиент протокола UDP. Является предком всех компонентов-клиентов, основанных на UDP

Таблица 5.1 *(продолжение)*

Компонент	Описание
IdDayTime	Клиент протокола DayTime (протокол получения даты и времени, RFC 867). Основан на TCP
IdDayTimeUDP	Клиент протокола DayTime, основанный на UDP
IdDNSResolver	Клиент DNS (службы имен доменов). Основан на UDP
IdEcho	Клиент протокола Echo (RFC 862). Основан на TCP
TIdEchoUDP	Клиент протокола Echo, основанный на UDP
IdFinger	Клиент протокола Finger (RFC 1288). Основан на TCP
IdFTP	Клиент протокола FTP (RFC 959). Основан на TCP
IdGopher	Клиент протокола Gopher (RFC 1436). Основан на TCP
IdHTTP	Клиент протокола HTTP (HTTP 1.0 — RFC 1945, HTTP 1.1 — RFC 2616). Основан на TCP
IdIcmpClient	Клиент протокола ICMP (RFC 972)
IdIdent	Клиент протокола идентификации (RFC 1413)
IdIMAP4Server	Клиент почтового протокола IMAP4 (RFC 2060). Основан на TCP
IdIPMCastClient	Клиент протокола IPM
IdIRC	Клиент протокола IRC (RFC 1459). Основан на TCP
IdLPR	Клиент службы печати
IdNNTP	Клиент протокола NNTP (RFC 977). Основан на TCP
IdPOP3	Клиент почтового протокола POP3 (RFC 1939). Основан на TCP
IdQOTD	Клиент протокола "Quote of the Day" (RFC 865). Основан на TCP
IdQOTDUDP	Клиент протокола "Quote of the Day", основанный на UDP
IdRexec	Клиент службы удаленного выполнения программ (Unix Remote Execution)
IdRSH	Клиент протокола RSH (Unix Remote Shell)
IdSMTP	Клиент почтового протокола SMTP (RFC 821, RFC 1869, RFC 2554). Основан на TCP.
IdSNPP	Клиент SNPP
IdSNTP	Клиент протокола SNTP (RFC 2030). Основан на UDP
IdSysLog	Клиент службы ведения системных журналов

Таблица 5.1 *(окончание)*

Компонент	Описание
IdTelnet	Клиент протокола Telnet. Основан на TCP
IdTime	Клиент протокола "Time Protocol" (RFC 868). Основан на TCP
IdTimeUDP	Клиент протокола "Time Protocol", основанный на UDP
IdTrivialFTP	Клиент протокола TFTP, основанного на UDP
IdWhois	Клиент протокола Whois (RFC 954). Основан на TCP

Хотя компоненты Indy предназначены для того, чтобы скрыть от программиста детали низкоуровневой реализации соответствующих протоколов, для эффективной работы с компонентом желательно знать основные принципы соответствующего интернет-протокола. Подробное описание различных протоколов Интернета выходит за рамки этой книги. Наиболее полным источником информации могут служить опубликованные в Интернете документы по стандартам международной сети Интернет под общим названием RFC (Request For Comment).

Компоненты-серверы Internet Direct реализуют большинство протоколов Интернета. Полный список компонентов-серверов Kylix 3 приводится в табл. 5.2.

Таблица 5.2. *Компоненты-серверы Indy*

Компонент	Описание
IdTCPServer	Базовый компонент-сервер протокола TCP. Является предком всех компонентов-серверов протоколов, основанных на TCP
IdUDPServer	Базовый компонент-сервер протокола UDP. Является предком всех компонентов-серверов протоколов, основанных на UDP
IdChargenServer	Сервер тестового протокола генерации символов (RFC 864). Основан на TCP
IdChargenUDPServer	Сервер протокола генерации символов, основанный на UDP
IdDayTimeServer	Сервер протокола DayTime (RFC 867). Основан на TCP
IdDayTimeUDPServer	Сервер протокола DayTime, основанный на UDP

Таблица 5.2 *(продолжение)*

Компонент	Описание
IdDICTServer	Сервер протокола словарей (RFC 2229). Основан на TCP
IdDISCARDServer	Сервер тестового протокола Discard (RFC 863). Основан на TCP
IdDiscardUDPServer	Сервер протокола Discard, основанный на UDP
IdEchoServer	Сервер протокола Echo (RFC 862). Основан на TCP
IdEchoUDPServer	Сервер протокола Echo, основанный на UDP
IdFingerServer	Сервер протокола Finger (RFC 1288). Основан на TCP
IdGopherServer	Сервер протокола Gopher (RFC 1436). Основан на TCP
IdHostNameServer	Сервер протокола HostName (RFC 953). Основан на TCP
IdHTTPServer	Сервер протокола HTTP (HTTP 1.0 — RFC 1945, HTTP 1.1 — RFC 2616). Основан на TCP
IdIdentServer	Сервер протокола идентификации (RFC 1413)
IdIMAP4Server	Сервер почтового протокола IMAP4 (RFC 2060). Основан на TCP
IdIPMCastServer	Сервер протокола IPM
IdIRCServer	Сервер протокола IRC (RFC 1459). Основан на TCP
IdMappedPortTCP	TCP-сервер, реализующий proxy-соединение
IdMappedPortUDP	UDP-сервер, реализующий proxy-соединение
IdMappedTelnet	Telnet-сервер, реализующий proxy-соединение
IdNNTPServer	Сервер протокола NNTP (RFC 977). Основан на TCP
IdQOTDServer	Сервер протокола "Quote of the Day" (RFC 865). Основан на TCP
IdQotdUDPServer	Сервер протокола "Quote of the Day", основанный на UDP
IdRexecServer	Сервер службы удаленного выполнения программ (Unix Remote Execution)
IdRSHServer	Сервер протокола RSH (Unix Remote Shell)
IdSimpleServer	Однопоточный сервер протокола TCP
IdSyslogServer	Сервер службы ведения системных журналов
IdTelnetServer	Сервер протокола Telnet. Основан на TCP

Таблица 5.2 *(окончание)*

Компонент	Описание
`IdTimeServer`	Сервер протокола "Time Protocol" (RFC 868). Основан на TCP
`IdTrivialFTPServer`	Сервер протокола TFTP, основанного на UDP
`IdTunnelMaster`, `IdTunnelSlave`	Многопоточные компоненты для работы с IP-тоннелями
`IdWhoisServer`	Сервер протокола Whois (RFC 954). Основан на TCP

Подробнее с различными серверами и клиентами Indy мы познакомимся далее в этой главе. Сейчас же будут рассмотрены несколько базовых классов Internet Direct, понимание работы которых позволит лучше понять принципы действия перечисленных выше компонентов.

Класс *TIdComponent*

Класс `TIdComponent` является основой практически всех компонентов Indy. Нам никогда не придется работать с этим классом непосредственно (если только мы сами не захотим написать Indy-совместимый компонент), однако этот класс вводит несколько полезных свойств и методов, которые используются при работе с классами-потомками `TIdComponent`.

Свойство `LocalName`, предназначенное только для чтения, содержит сетевое имя локальной машины. (Для получения имени это свойство использует функцию `WSGetHostName`).

Событие `OnStatus` вызывается всякий раз при изменении состояния соединения. Обработчику события передается параметр типа `TIdStatus`, идентифицирующий состояние соединения, и строка с текстом, описывающим состояние. Событие `OnStatus` можно использовать в приложении для информирования пользователя об установке или разрыве соединения, успешном или неудачном поиске узла и т. п.

Класс *TIdThread*

В предыдущей главе говорилось о том, что основой архитектуры Indy является многопоточность. В связи с этим большую роль в Indy-приложениях играет класс `TIdThread`, являющийся наследником класса `TThread`. С точки зрения сервера класс `TIdThread` не только позволяет распараллелить обработку запросов, но идентифицирует каждый конкретный запрос. В Indy-приложениях нам не придется явным образом создавать объекты-потоки.

Компоненты-серверы самостоятельно управляют этими объектами. Однако иногда нам придется обращаться к отдельным свойствам и методам объектов класса `TIdThread`. Далее приводится перечень свойств и методов класса `TIdThread`.

- `Data` — свойство типа `TObject`. Ссылка на объект, хранящий данные, специфичные для данного экземпляра `TIdThread`. Мы можем использовать это свойство для связывания потока с произвольными данными, необходимыми нашему приложению.
- `StopMode` — свойство типа `TIdThreadStopMode`. Определяет действие, которое будет предпринято при вызове метода `Stop`: `smTerminate` — поток завершается; `smSuspend` — поток временно приостанавливается.
- `Stopped` — свойство типа `Boolean`. Принимает значение `True`, если поток приостановлен. Это свойство лишь информирует о состоянии потока, но не управляет им.
- `TerminatingException` — свойство типа String. Позволяет указать текстовое сообщение, выводимое в случае возникновения исключительной ситуации в теле потока.
- `Start` — процедура. Выполняет запуск/возобновление потока.
- `Stop` — процедура. Приостанавливает выполнение потока.
- `Synchronize` — процедура. Синхронизация методов. Позволяет вызвать методы объекта потока в безопасном режиме, т. е. не нарушая работу других потоков.

Класс *TIdTCPConnection*

Другой важный элемент архитектуры Indy — класс `TIdTCPConnection`. Этот класс инкапсулирует все свойства отдельного TCP-соединения и является основой всех TCP-клиентов Indy, а также класса `TIdSimpleServer`. Поскольку протокол TCP создает постоянные соединения и требует сохранять данные о состоянии соединения, в данный момент времени в системе должно быть столько экземпляров объекта `TIdTCPConnection`, сколько соединений установлено в данный момент.

Класс `TIdTCPConnection` предоставляет методы, свойства и события, позволяющие выполнять практически любые операции с TCP соединением и получать полную информацию о его параметрах и текущем состоянии. Со многими методами класса `TIdTCPConnection` мы познакомимся в следующих разделах этой главы, здесь же рассмотрим некоторые из них.

Свойство `Socket` позволяет обращаться непосредственно к функциям сетевого интерфейса. Методы класса `TIdIOHandlerSocket`, объектом которого

является свойство Socket, инкапсулируют практически все функции для работы с сетевыми сокетами.

К методам записи данных в TCP-канал относятся следующие:

- метод Write — запись значения переменной типа String;
- метод WriteBuffer — запись содержимого блока памяти;
- метод WriteFile — пересылка содержимого файла, заданного именем;
- метод WriteInteger — запись целочисленного значения;
- метод WriteLn — запись строки с добавлением символа конца строки.

Особую роль играют перегруженные методы SendCmd. Эти методы позволяют посылать серверу команды высокоуровневых протоколов, основанных на протоколе TCP. Указанные методы не просто посылают данные, но и позволяют проверить, соответствует ли ответ сервера ожидаемому. Обычно в ответ на команду сервер возвращает статус-код. При вызове метода SendCmd нужно указывать список допустимых статус-кодов, ожидаемых от сервера. Если сервер возвращает статус-код, отсутствующий в списке, вызывается исключение.

Далее перечислены основные методы класса TIdTCPConnection, позволяющие считывать данные из канала.

- ReadBuffer — чтение данных в область памяти, заданную указателем.
- ReadInteger — чтение целочисленного значения.
- ReadLn — считывание из канала данных в переменную типа String. Чтение данных выполняется до тех пор, пока из потока не будет считан один из символов конца строки: #0, #10, #13 или #13#10.
- ReadStream — чтение данных в объект-поток.
- ReadStrings — чтение данных в объект типа TStrings.

Методы этой группы являются блокирующими, но позволяют установить максимальное время ожидания (timeout) с помощью свойства ReadTimeout. Если время ожидания не установлено бесконечным, метод ReadLn вернет управление либо после удачного завершения соответствующей операции, либо по истечении указанного времени. Если до истечения указанного времени сервер не получит данных, метод чтения данных вернет пустое значение. При этом свойство ReadLnTimedOut примет значение true.

Важную роль в работе TCP-клиентов и серверов играют методы CheckForDisconnect и CheckForGracefulDisconnect. Эти методы позволяют проверить, не было ли соединение разорвано удаленным приложением. По моим наблюдениям метод CheckForGracefulDisconnect является более надежным. По умолчанию методы сконфигурированы так, что в случае разрыва соединение вызывается исключительная ситуация. Эти методы также могут вызывать событие OnDisconnected.

Полезной функцией класса `TIdTCPConnection` является также метод `CheckResponse`, извлекающий численные значения статус-кодов из стандартных ответов TCP-серверов. Благодаря этому методу очень легко организовать работу TCP-клиента по общепринятой схеме конечного автомата. Для проверки состояния соединения можно использовать метод `Connected`, представляющий собой функцию, возвращающую значение типа `Boolean`: `false` или `true`.

Поскольку в объектной модели Indy класс `TIdTCPConnection` является низкоуровневым, в нем отсутствует метод установки соединения (соединение устанавливается компонентом-клиентом, использующим класс `TIdTCPConnection`), но есть методы `Disconnect` и `DisconnectSocket`, позволяющие разорвать соединение.

Класс *TIdUDPBase*

Хотя большинство высокоуровневых протоколов, с которыми Indy имеет дело, основаны на транспортном протоколе TCP, протокол UDP также активно используется. Класс `TIdUDPBase` играет для UDP-компонентов ту же роль, что класс `TIdTCPConnection` для TCP-компонентов. К наиболее важным методам класса `TIdUDPBase` относятся:

- метод `Send`, позволяющий отправить UDP-пакет на определенный порт определенного узла сети;
- метод `Broadcast`, позволяющий отправить UDP-пакет на определенный порт всех узлов сети (обычно сетевые шлюзы и маршрутизаторы позволяют распространять "широковещательные" UDP-пакеты лишь в рамках локальной сети);
- перегруженные методы `ReceiveBuffer`, позволяющие считывать UDP-данные в выделенный буфер;
- перегруженные методы `RecieveString`, позволяющие считывать UDP-данные в переменную типа `String`.

Среди наиболее важных свойств класса `TIdUDPBase` отметим свойство `Binding`, позволяющее получить данные о сокете, используемом для отправки и приема сообщений.

Простая модель клиент-сервер на основе протокола TCP

Отличительной особенностью протокола TCP является установка виртуального канала связи между клиентом и сервером. В отличие от протокола UDP протокол TCP позволяет отслеживать доставку отдельных пакетов и требовать повторную отправку утерянных пакетов. Сравнительная простота и высокая надежность протокола TCP является причиной того, что большинство интернет-протоколов основаны на протоколе TCP.

Многие протоколы, основанные на TCP, достаточно сложны. Вместе с тем протокол TCP можно использовать для простого обмена данными между клиентами и серверами.

В этом разделе мы напишем два приложения — клиент и сервер, использующие протокол TCP для обмена данными. Для этого мы воспользуемся компонентами `IdSimpleServer` и `IdTCPClient`. Компонент `IdSimpleServer` представляет собой простой однопоточный сервер, способный обрабатывать в каждый данный момент времени только одно соединение. Обмен данными выполняется в синхронном режиме (при помощи блокирующих функций), что упрощает использование компонента. Простой TCP-сервер лучше всего подходит для организации обмена данными между двумя удаленными процессами.

Для того чтобы сервер мог функционировать, необходимо присвоить значения свойствам `BoundIP` и `BoundPort` компонента `IdSimpleServer`. Сервер приводится в состояние ожидания соединения при помощи вызова блокирующего метода `Listen`. Этот метод представляет собой функцию, возвращающую значение типа `Boolean`. Функция `Listen` возвращает управление только после инициализации соединения удаленным клиентом. При этом функция возвращает значение `true`, если соединение было установлено успешно, и `false` в противном случае. После того как соединение установлено, мы можем воспользоваться методами `ReadLn`, `ReadString` и им подобными для чтения данных, переданных клиентом и методами `Write`, `WriteBuffer` и т. п. для передачи данных. Указанные методы унаследованы классами `TIdSimpleServer` и `TIdTCPClient` от общего предка — класса `TIdTCPConnection`, который является предком как компонента `IdSimpleServer`, так и компонента `IdTCPClient`, так что оба компонента могут использовать одну и ту же модель ввода-вывода. После завершения сеанса связи следует вызвать метод `EndListen`, который освободит прослушиваемый порт.

Важнейшими свойствами компонента `IdTCPClient` являются `Host` и `Port`, указывающие соответственно адрес и порт сервера, с которым требуется установить соединение. После этого можно вызывать метод `Connect`, инициализирующий соединение. Метод `Connect` возвращает управление после установки соединения. Если в процессе установки соединения возникла ошибка, вызывается соответствующее исключение.

Для разрыва соединения можно воспользоваться методом `Disconnect`. При вызове этого метода вызывается событие `OnDisconnected`. Для "быстрого и грязного" завершения соединения (например, в случае возникновения критической ошибки) можно воспользоваться методом `DisconnectSocket`.

Наименования, присвоенные в Indy свойствам, хранящим значения портов, могут вызвать некоторую путаницу. При запуске сервера он начинает прослушивание определенного порта в ожидании запроса на соединение. К этому же порту должен обращаться клиент, устанавливающий соедине-

ние. После того, как сервер получает от клиента запрос на установку соединения, между клиентом и сервером устанавливается новое соединение, для которого как на стороне сервера, так и на стороне клиента открываются новые порты. Такой механизм похож на принцип действия многоканального телефона и позволяет серверу принимать множество запросов на соединение на один порт. Номер порта, прослушиваемого сервером, задается свойством `BoundPort` компонента `IdSimpleServer`. Для компонента-клиента этот же номер должен быть присвоен свойству `Port`, а свойство `BoundPort` выполняет в компоненте `IdTCPClient` другие функции. Обычно новый порт, открываемый клиентом после того, как сервер разрешил соединение, выделяется автоматически из списка свободных портов системы. Компонент `IdTCPClient` может использовать свойство `BoundPort` для указания фиксированного номера порта для установки соединения. Можно также использовать свойства `BoundPortMin` и `BoundPortMax` для указания диапазона, из которого должен быть выделен новый порт. Как правило, свойства клиента `BoundPort`, `BoundPortMin` и `BoundPortMax` используются при тестировании сетевой подсистемы. Необходимо помнить лишь о том, что свойству `BoundPort` сервера соответствует свойство клиента `Port`.

Метод `Connect` является блокирующим, но позволяет установить максимальное время ожидания (timeout), которое задается параметром `ATimeOut`. Проиллюстрируем все вышесказанное практическими примерами. В листинге 5.1 приводится исходный текст простого TCP-сервера. Кроме компонента `IdSimpleServer` наше приложение-сервер содержит компонент `TMemo` (`LogMemo`) для записи данных об информации, переданной клиентом, и кнопку `TButton` (`ConnectButton`), которая запускает сервер. Для того чтобы методы `Listen` и `ReadLn` не "замораживали" сервер, в его главную форму следует добавить компонент `IdAntiFreeze`.

Листинг 5.1. Простой TCP-сервер

```
//---------------------------------------------------------------------

#include <clx.h>
#pragma hdrstop

#include "Main.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//---------------------------------------------------------------------
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------

void __fastcall TForm1::IdSimpleServer1Disconnected(TObject *Sender)
{
  LogMemo->Lines->Add("Disconnected");
  IdSimpleServer1->EndListen();
  ConnectButton->Enabled = false;
}
//---------------------------------------------------------------------

void __fastcall TForm1::ConnectButtonClick(TObject *Sender)
{
  ConnectButton->Enabled = false;
  if IdSimpleServer1->Listen();
  {
    try
    {
      while (true)
      {
        String S = IdSimpleServer1->ReadLn();
        LogMemo->Lines->Add("Client sent : "+S);
        IdSimpleServer1->WriteLn("OK, "+S);
        IdSimpleServer1->CheckForGracefulDisconnect();
      }
    }
    catch (EIdException *e)
    {
    }
  }
  else
  {
    ConnectButton->Enabled = true;
  }
}
```

```
//---------------------------------------------------------------------------
TForm1::FromClose(TObject *Sender)
{
  IdSimpleServer1->EndListen();
}
```

Для краткости мы опустили код заголовочного файла Main.h. Кнопка **Connect** переводит сервер в состояние ожидания соединения. Если соединение установлено (`Listen` вернул `true`) программа входит в цикл чтения данных. Получая от клиента строку, сервер возвращает эту строку с прибавлением "OK". Выход из цикла происходит при разрыве соединения с клиентом, так как при этом `CheckForGracefulDisconnect` вызывает исключительную ситуацию. Обработчик события `OnDisconnected` вызывает метод `EndListen`, сбрасывающий внутренние переменные состояния сервера. Для того чтобы вернуть сервер в состояние ожидания соединения, необходимо снова нажать кнопку **Connect**.

Теперь напишем программу-клиент для работы с сервером. Клиент (листинг 5.2) устанавливает связь с сервером (компонент-кнопка `ConnectButton`), посылает серверу строку `Edit1->Text`, считывает ответ сервера. Разрыв соединения выполняется при помощи компонента `DisconnectButton`.

Листинг 5.2. Программа-клиент

```
//---------------------------------------------------------------------------

#include <clx.h>
#pragma hdrstop

#include "Main.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------

void __fastcall TForm1::SendButtonClick(TObject *Sender)
```

```
{
  try
  {
    String S = (String)Edit1->Text;
    IdTCPClient1->WriteLn(S);
    S = IdTCPClient1->ReadLn();
    Label1->Caption = S;
    IdTCPClient1->CheckForGracefulDisconnect();
  }
  catch (EIdException *e)
  {
    ShowMessage(e->Message);
    IdTCPClient1->Disconnect();
  }
}
//---------------------------------------------------------------------------

void __fastcall TForm1::DisconnectButtonClick(TObject *Sender)
{
  IdTCPClient1->Disconnect();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::ConnectButtonClick(TObject *Sender)
{
  IdTCPClient1->Connect();
  DisconnectButton->Enabled = true;
  ConnectButton->Enabled = false;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::IdTCPClient1Disconnected(TObject *Sender)
{
  DisconnectButton->Enabled = false;
  ConnectButton->Enabled = true;
}
//---------------------------------------------------------------------------
```

Для того, чтобы клиент и сервер могли взаимодействовать, значение свойства `BoundIP` сервера должно соответствовать значению свойства `Host` клиента, а значение свойства `BoundPort` сервера — значению `Port` клиента. На рис. 5.1 показано работающее приложение-сервер.

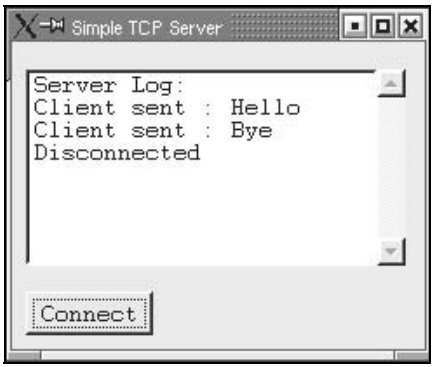


Рис. 5.1. Простой TCP-сервер

Следует отметить одну особенность нашей простейшей клиент-серверной системы. При разрыве соединения со стороны сервера, клиент не получает автоматического уведомления об этом. Установить, что соединение разорвано можно только после очередной попытки отправки данных.

В следующих разделах этой главы мы рассмотрим более сложные (и эффективные) методы работы с клиентами и серверами Indy.

Компонент *IdTCPServer*

Компонент `IdTCPServer` позволяет создавать полноценные серверы на основе протокола TCP. В отличие от сервера, рассмотренного в предыдущем разделе, этот компонент реализует многопоточную обработку запросов и, следовательно, способен поддерживать несколько одновременных соединений. Класс `TIdTCPServer` является предком всех компонентов-серверов Indy, инкапсулирующих протоколы TCP, поэтому понимание принципов его работы необходимо для понимания принципов работы остальных TCP-серверов. Класс `TIdTCPServer` обладает множеством свойств и методов, полное описание которых можно найти в документации Indy. В этом разделе мы ограничимся рассмотрением наиболее важных элементов.

Для связывания с определенными адресами и портами класс `TIdTCPServer`, как и все его потомки, использует свойство `Bindings` типа `TIdSocketHandlers`. Тип `TIdSocketHandlers` представляет собой класс-коллекцию элементов `TIdSocketHandler`. Каждый элемент этой коллекции, кроме всего прочего, содержит пару IP-адрес-порт, с которой должен быть связан сервер. Таким образом,

как и всякий полноценный сервер, компонент `IdTCPServer` может быть связан с несколькими IP-адресами и портами одновременно.

Запуском и остановкой сервера управляет свойство `Active`. Сервер принимает поступающие от клиентов запросы на установку соединения, только если свойству `Active` присвоено значение `true`.

Поскольку сервер `TIdTCPServer` позволяет устанавливать несколько соединений одновременно, в нем предусмотрена возможность ограничения максимально-допустимого числа одновременных соединений. Это число задается свойством `MaxConnections`. Присвоение свойству `MaxConnections` значения 0 делает возможное число одновременных соединений неограниченным.

Класс `TIdTCPServer` и все его потомки позволяют применять два подхода к обработке соединений. Первый подход основан на обработчиках событий компонента-сервера. Каждое соединение между сервером и клиентом выполняется в отдельном потоке *(см. разд. "Знакомство с компонентами Internet Direct" гл. 4)*. Потоки вызывают обработчики событий компонента-сервера, передавая им в качестве параметра ссылку на самих себя. Рассмотрим тип процедуры-обработчика события `OnConnect`, которое возникает при каждой попытке установления соединения:

```
IdTCPServer1Connect(AThread: TIdPeerThread);
```

Параметр `AThread` представляет собой ссылку на поток, созданный для обработки соединения, вызвавшего событие, и инкапсулированный в объекте класса `TIdPeerThread`. Класс `TIdPeerThread` является контейнером потоков соединений для всех многопоточных компонентов Indy, работающих с протоколом TCP. К методам и свойствам своего предка — класса `TIdThread` — класс `TIdPeerThread` добавляет новое свойство — `Connection`. Свойство `Connection`, представляющее собой объект класса `TIdTCPServerConnection`, позволяет обработчику события получить всю необходимую информацию о конкретном соединении.

Иерархические отношения между классами Indy, инкапсулирующими соединения, показаны на рис. 5.2.

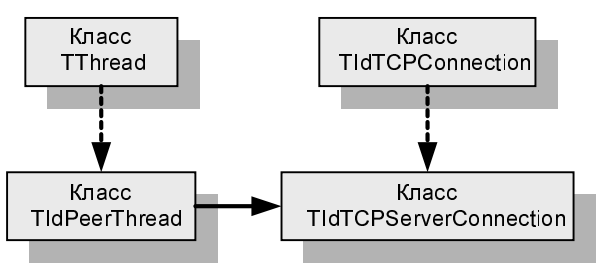


Рис. 5.2. Классы Indy, инкапсулирующие соединения

Как видно из схемы (рис. 5.2), обработчик события компонента-сервера может получить доступ к объекту класса `TIdTCPConnection`, а следовательно, прием, передачу данных и другие операции с соединением можно осуществлять теми же методами, что и в случае однопоточного сервера.

Важную роль в обработке соединений указанным способом играет событие `OnExecute`. Если созданное TCP-соединение не активно, соответствующий поток находится в режиме приостановки. Выполнение потока возобновляется всякий раз, когда в TCP-канале происходит нечто, требующее реакции со стороны сервера. В этот момент и вызывается обработчик события `OnExecute`, получающий ссылку на экземпляр объекта соответствующего потока.

Другой подход к обработке запросов клиента основан на обработчиках команд. В рамках большинства TCP-протоколов (характерным примером является протокол HTTP) клиент посылает серверу команды, на которые сервер отвечает. Новая архитектура класса `TIdTCPServer` тесно связана с этой особенностью TCP-протоколов.

Теперь у класса `TIdTCPServer` есть свойство `CommandHandlers`, представляющее собой коллекцию объектов `TIdCommandHandler`. Каждый объект `TIdCommandHandler` инкапсулирует обработчик определенной команды. Для того чтобы задействовать обработчики `CommandHandlers`, необходимо присвоить значение `true` свойству `CommandHandlersEnabled` класса `TIdTCPServer`. В этом режиме класс `TIdTCPServer` выступает в роли диспетчера команд. Каждый раз, когда TCP-клиент посылает серверу команду, сервер находит соответствующий обработчик `IdCommandHandler` и вызывает его событие `OnCommand`. Команда, обрабатываемая данным обработчиком, указывается в свойстве `Command` объекта `TIdCommandHandler`.

Впрочем, включение механизма обработчиков команд не отменяет описанного выше механизма событий. Кроме уже известных нам событий класса `TIdTCPServer` `OnConnect`, `OnDisconnect` и `OnExecute`, введены новые события:

- `OnBeforeCommandHandler` — вызывается перед вызовом обработчика команды;
- `OnAfterCommandHandler` — вызывается после вызова обработчика команды;
- `OnNoCommandHandler` — вызывается в том случае, если для поступившей команды не найден обработчик.

Последнее событие представляет особый интерес, так как позволяет обрабатывать некорректные ситуации, возникающие в процессе TCP-соединения.

Рассмотрим обработчик события `OnCommand` класса `TIdCommandHandler`:

```
TIdCommandEvent = procedure (ASender: TIdCommand) of object;
```

В качестве параметра обработчику передается ссылка на объект класса `TIdCommand`, который содержит данные и о соединении, по которому была

передана команда, и о самой команде. Многие команды TCP-протоколов передаются со списком параметров. Объект TIdCommand хранит параметры команды в свойстве Params типа TStrings. Для того, чтобы список параметров был разобран правильно, необходимо присвоить соответствующие требованиям протокола значения свойствам CmdDelimiter и ParamDelimiter объекта TIdCommandHandler и установить свойство ParseParams этого объекта, равным true. Доступ к необработанному списку параметров команды можно получить при помощи свойства UnparsedParams.

Свойство Thread объекта TIdCommand содержит ссылку на экземпляр объекта-потока TIdPeerThread, инкапсулирующего соответствующее TCP-соединение, а свойство CommandHandler ссылается на объект TIdCommandHandler, являющийся обработчиком данной команды.

Одной из причин, которая может заставить нас обратиться к объекту-обработчику из обработчика события OnCommand, является свойство Disconnect объекта TIdCommandHandler. Это свойство типа Boolean определяет, будет ли TCP-соединение разорвано после обработки поступившей команды или нет. Для некоторых команд протокола этот вопрос решается однозначно. Пусть мы разрабатываем некий собственный TCP-протокол, в котором определена команда CLOSE, завершающая соединение. Очевидно, что после выполнения команды CLOSE соединение должно быть разорвано в любом случае. Мы можем присвоить свойству Disconnect объекта-обработчика команды CLOSE значение true еще во время разработки программы, и нам никогда не понадобится его менять. Однако многие команды работают по-другому. Допустим, в нашем протоколе есть также команда AUTH, выполняющая авторизацию удаленного клиента. В качестве параметров команде AUTH передаются имя клиента и пароль. Обработчик команды сравнивает переданные реквизиты удаленного клиента с теми, что содержатся в его базе учетных записей клиентов, и на основании этого сравнения либо разрешает клиенту дальнейшую работу с сервером, либо нет. При этом вполне логично, что в случае успешной авторизации клиента TCP-соединение сохраняется, а в случае неудачи — разрывается. Таким образом, в этом случае решение о том, разрывать ли соединение после выполнения TCP-команды, зависит от параметров самой команды и должно приниматься во время выполнения программы.

Разрыв соединения можно выполнить при помощи метода Disconnect объекта TIdTCPConnection, но этого не следует делать в обработчике OnCommand, так как в этом случае сервер не успеет отправить сформированный ответ. Если мы хотим разорвать соединение при помощи непосредственного обращения к методам Disconnect и DisconnectSocket, это лучше всего делать в обработчике события OnAfterCommandHandler. Обработчик OnAfterCommandHandler получает ссылку на объект TIdPeerThread, которую перед этим получил обработчик OnCommand объекта TIdCommandHandler. Мы

можем присвоить свойству `Data` объекта-потока значение, указывающее, что соединение нужно закрыть. Обработчик `OnAfterCommandHandler` прочтет это значение и выполнит соответствующую операцию.

Другой, более простой способ решения этой проблемы заключается в изменении значения свойства `Disconnect` объекта `TIdCommandHandler`, на который ссылается свойство `CommandHandler` объекта `TIdCommand`. При этом нужно помнить только, что для всех TCP-соединений используется один и тот же экземпляр объекта `TIdCommandHandler`, так что обработчики соединений, использующие свойство `Disconnect` для управления разрывом соединения, не должны делать никаких предположений относительно состояния этого свойства на момент вызова обработчика.

Ответ сервера формируется при помощи свойств `Reply` и `Response` объекта `IdCommand`. Свойство `Reply` является ссылкой на объект класса `TIdRFCReply`, который позволяет сформировать стандартный ответ сервера по правилам, принятым для протоколов, основанных на TCP. Этот объект позволяет указать строку, возвращаемую сервером, а также численное значение статус-кода.

Свойство `Response` представляет собой ссылку на объект класса `TStrings`. Очевидно, что с помощью этого свойства можно передавать "многострочный" ответ сервера — обычно это данные, запрошенные клиентом.

Данные `Response` передаются сервером автоматически, тогда как для отправки клиенту значения `Reply` следует использовать метод `SendReply` объекта `IdCommand`. Метод `SendReply` всегда передает значение свойства `Reply` до того, как будет передано содержимое свойства `Response`, но делает это только в том случае, если свойству `PerformReply` объекта `IdCommand` было присвоено значение `true`.

На рис. 5.3 показана структура ответа сервера с использованием свойств `Reply` и `Response`.

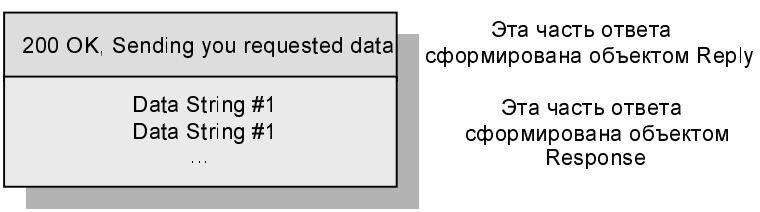


Рис. 5.3. Структура ответного сообщения TCP-сервера

В качестве примера использования компонента `IdTCPServer` напишем приложение-сервер и приложение-клиент, реализующие простой протокол на основе TCP. Наш TCP-протокол включает четыре команды.

- `HELP` — получение сведений о командах протокола. Команда не требует авторизации, после ее выполнения сервер разрывает соединение.

- `AUTH user_name password` — авторизация клиента. Если авторизация прошла успешно, сервер возвращает статус-код 200 и сообщение "You are logged in", соединение при этом не разрывается. Если клиент не прошел авторизацию, сервер возвращает статус-код 500 и сообщение "Login failed", и разрывает соединение.
- `HELLO` – команда без параметров. Требует авторизации. Если клиент, установивший соединение, прошел авторизацию, сервер возвращает статус-код 200 и сообщение "Hello user_name", сохраняя соединение. В противном случае (если клиент не был предварительно авторизован), сервер возвращает статус-код 500 и сообщение "You are not logged in", и разрывает соединение.
- `CLOSE` – команда завершения сеанса. Если клиент, установивший соединение, прошел авторизацию, сервер возвращает статус-код 200 и сообщение "Bye, user_name", в противном случае сервер возвращает статус-код 500 и сообщение "You are not logged in". Соединение разрывается в любом случае.

В качестве признака окончания передачи данных используется двойной символ `#10`.

Для обработки команд наш сервер использует объекты `IdCommandHandler`.

Теперь приступим к написанию приложения-сервера.

1. Создадим новый проект в Kylix Delphi IDE.
2. Разместим в главной форме компонент `IdTCPServer`.
3. Щелкним мышью в поле `CommandHandlers` инспектора объектов. При этом будет открыто окно редактора обработчиков команд.
4. В этом окне следует создать четыре обработчика для команд, перечисленных выше.
5. Полю `Command` каждого обработчика присвоим имя соответствующей команды. Для удобства можно назначить объектам-обработчикам имена, совпадающие с именами команд.

На рис. 5.4 показан внешний вид окна редактора после создания всех обработчиков.

1. Полям `Disconnect` обработчиков команд `HELP` и `CLOSE` присвоим значение `true`.
2. Добавим в форму компонент `TButton` для запуска/останова сервера (назовем компонент `StartButton`) и компонент `TMemo` для записи отчета о событиях сервера (назовем этот компонент `LogMemo`).

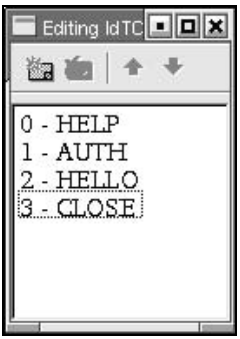


Рис. 5.4. Окно редактора обработчиков команд

Для идентификации удаленных клиентов мы воспользуемся компонентом `IdUserManager`, расположенным на вкладке Indy Misc. Этот компонент позволяет хранить список учетных записей пользователей с их именами и паролями, а также дополнительную информацию о правах пользователя. Компонент `TIdUserManager` используется компонентом `TIdFTPServer` для авторизации удаленных пользователей FTP.

Доступ к списку учетных записей пользователей можно получить с помощью свойства `Accounts` класса `TIdUserManager`. Редактировать список учетных записей можно как на этапе разработки, так и во время выполнения программы. С помощью компонента `IdUserManager` можно осуществлять также идентификацию и авторизацию пользователей (метод `AuthenticateUser`). Перед тем как запустить сервер, нам следует создать хотя бы одну учетную запись с именем и паролем удаленного клиента.

Остальное программирование нашего сервера сводится к написанию обработчиков событий компонента `IdTCPServer` и обработчиков событий `OnCommand` объектов-обработчиков команд.

Исходный текст главного модуля программы-сервера приводится в листинге 5.3.

Листинг 5.3. Программа-сервер простого TCP-протокола

```
unit Main;

interface

uses
  SysUtils, Types, Classes, Variants, QTypes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls, IdBaseComponent, IdComponent, IdTCPServer,
  IdUserAccounts, IdServerIOHandler, IdSSLOpenSSL, IdIOHandler,
  IdIOHandlerSocket;
```

```
type
  TForm1 = class(TForm)
    IdTCPServer1: TIdTCPServer;
    IdUserManager1: TIdUserManager;
    Button1: TButton;
    Memo1: TMemo;
    procedure IdTCPServer1HELPCommand(ASender: TIdCommand);
    procedure IdTCPServer1AUTHCommand(ASender: TIdCommand);
    procedure IdTCPServer1HELLOCommand(ASender: TIdCommand);
    procedure IdTCPServer1CLOSECommand(ASender: TIdCommand);
    procedure Button1Click(Sender: TObject);
    procedure IdTCPServer1Connect(AThread: TIdPeerThread);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure IdTCPServer1Disconnect(AThread: TIdPeerThread);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.IdTCPServer1HELPCommand(ASender: TIdCommand);
begin
  ASender.Response.Add('Simple TCP Server');
  ASender.Response.Add('Supported commands:');
  ASender.Response.Add('AUTH user_name password');
  ASender.Response.Add('HELLO');
  ASender.Response.Add('CLOSE'+#10+#10);
  Asender.Reply.SetReply(200,'OK');
  Asender.SendReply;
end;
```

```
procedure TForm1.IdTCPServer1AUTHCommand(ASender: TIdCommand);
var
  SL : TStringList;
begin
  if idUserManager1.AuthenticateUser(ASender.Params.Strings[0],
  ASender.Params.Strings[1]) then
  begin
    SL := TStringList.Create();
    SL.Add(ASender.Params.Strings[0]);
    SL.Add(ASender.Params.Strings[1]);
    Asender.Thread.Data := SL;
    ASender.Response.Add('You are logged in!'+#10+#10);
    ASender.CommandHandler.Disconnect := False;
    Asender.Reply.SetReply(200,'OK');
    Asender.SendReply;
  end else
  begin
    ASender.Response.Add('Login Failed'+#10+#10);
    ASender.CommandHandler.Disconnect := True;
    ASender.Thread.Data := nil;
    ASender.Reply.SetReply(500,'Failure');
    ASender.SendReply;
  end;
end;

procedure TForm1.IdTCPServer1HELLOCommand(ASender: TIdCommand);
var
  SL : TStringList;
begin
  ASender.Response.Clear;
  if Asender.Thread.Data = nil then
  begin
    ASender.Response.Add('You are not logged in!'+#10+#10);
    ASender.CommandHandler.Disconnect := True;
    Asender.Reply.SetReply(500,'Failure');
    Asender.SendReply;
  end else
  begin
```

```
    SL := Asender.Thread.Data as TStringList;
    ASender.Response.Add('Hello '+SL.Strings[0]+#10+#10);
    ASender.CommandHandler.Disconnect := False;
    ASender.Reply.SetReply(200,'OK');
    ASender.SendReply;
  end;
end;

procedure TForm1.IdTCPServer1CLOSECommand(ASender: TIdCommand);
var
  SL : TStringList;
begin
  if Asender.Thread.Data = nil then
  begin
    ASender.Response.Add('You are not logged in!'+#10+#10);
    Asender.Reply.SetReply(500,'Failure');
    Asender.SendReply;
  end else
  begin
    SL := Asender.Thread.Data as TStringList;
    ASender.Response.Add('Bye, '+SL.Strings[0]+#10+#10);
    ASender.Reply.SetReply(200,'OK');
    ASender.SendReply;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
try
  IdTCPServer1.Active := not IdTCPServer1.Active;
  if IdTCPServer1.Active then Button1.Caption := 'Stop'
  else Button1.Caption := 'Start';
except
  ShowMessage('Exception!');
end;
end;

procedure TForm1.IdTCPServer1Connect(AThread: TIdPeerThread);
```

```
begin
  Memo1.Lines.Add('Connected!');
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  IdTCPServer1.Active := False;
end;

procedure TForm1.IdTCPServer1Disconnect(AThread: TIdPeerThread);
begin
  Memo1.Lines.Add('Disconnected!');
end;
end.
```

В процессе обработки команд нам необходимо знать, прошел ли клиент авторизацию или нет. Для этого мы используем объект класса `TStringList`. Если авторизация клиента прошла успешно, в объект `TStringList` записываются имя и пароль клиента, и ссылка на объект присваивается свойству `Data` объекта, инкапсулирующего поток обработки соединения. Обработчики команд `HELLO` и `CLOSE` проверяют значение свойства `Data` и таким образом определяют, прошел ли клиент авторизацию.

Напишем теперь приложение-клиент для нашего протокола. Движущей силой нашего приложения-клиента является компонент `IdTCPClient`. На рис. 5.5 показана форма приложения-клиента в рабочем состоянии.

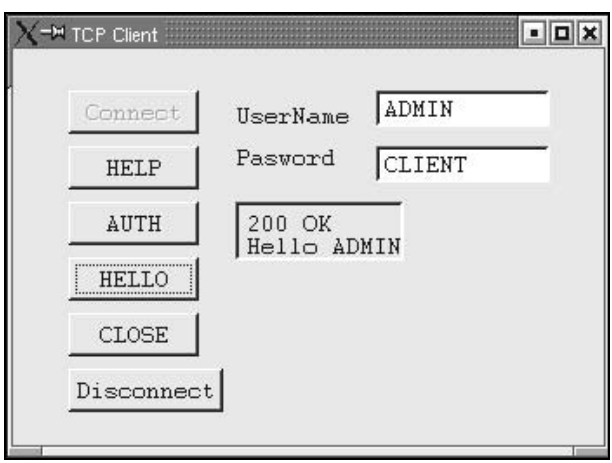


Рис. 5.5. Приложение-клиент протокола TCP

Наше приложение-клиент содержит кнопки для установления и разрыва соединения с сервером, а также кнопки для отправки команд серверу. Кроме этого в приложении предусмотрены поля ввода имени клиента и пароля для авторизации и текстовое поле, в котором можно просматривать ответ сервера.

Исходный текст главного модуля приложения-клиента приводится в листинге 5.4.

Листинг 5.4. Приложение-клиент

```
unit Main;

interface

uses
  SysUtils, Types, Classes, Variants, QTypes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls, IdBaseComponent, IdComponent,
  IdTCPConnection, IdTCPClient;

type
  TForm1 = class(TForm)
    IdTCPClient1: TIdTCPClient;
    ConnectButton: TButton;
    UserName: TEdit;
    Password: TEdit;
    AuthButton: TButton;
    Label1: TLabel;
    DisconnectButton: TButton;
    HelloButton: TButton;
    CloseButton: TButton;
    HelpButton: TButton;
    Label2: TLabel;
    Label3: TLabel;
    procedure ConnectButtonClick(Sender: TObject);
    procedure AuthButtonClick(Sender: TObject);
    procedure DisconnectButtonClick(Sender: TObject);
    procedure HelloButtonClick(Sender: TObject);
    procedure CloseButtonClick(Sender: TObject);
    procedure HelpButtonClick(Sender: TObject);
```

```
    procedure IdTCPClient1Disconnected(Sender: TObject);
    procedure IdTCPClient1Connected(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    function SendCommand(const Cmd : String) : String;
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

function TForm1.SendCommand(const Cmd : String) : String;
begin
  try
    IdTCPClient1.SendCmd(Cmd);
    Result := IdTCPClient1.ReadLn(#10+#10);
    IdTCPClient1.CheckForGracefulDisconnect();
  except
    IdTCPClient1.Disconnect;
  end;
end;

procedure TForm1.ConnectButtonClick(Sender: TObject);
begin
  IdTCPClient1.Connect();
end;

procedure TForm1.AuthButtonClick(Sender: TObject);
begin
  Label1.Caption := SendCommand('AUTH '+UserName.Text+' '+Password.Text);
end;

procedure TForm1.DisconnectButtonClick(Sender: TObject);
```

```
begin
  IdTCPClient1.Disconnect();
end;

procedure TForm1.HelloButtonClick(Sender: TObject);
begin
  Label1.Caption := SendCommand('HELLO');
end;

procedure TForm1.CloseButtonClick(Sender: TObject);
begin
  Label1.Caption := SendCommand('CLOSE');
end;

procedure TForm1.HelpButtonClick(Sender: TObject);
begin
  Label1.Caption := SendCommand('HELP');
end;

procedure TForm1.IdTCPClient1Disconnected(Sender: TObject);
begin
  ConnectButton.Enabled := True;
  DisconnectButton.Enabled := False;
end;

procedure TForm1.IdTCPClient1Connected(Sender: TObject);
begin
  ConnectButton.Enabled := False;
  DisconnectButton.Enabled := True;
end;

end.
```

Для передачи команды серверу используется метод `SendCmd` базового класса `TIdTCPConnection`. В качестве параметра этому методу передается строка, содержащая команду и ее параметры. Метод `SendCmd` работает в синхронном режиме и возвращает строку, содержащую ответ сервера.

Другие TCP-серверы Indy, протокол UDP

Компоненты-серверы Indy, реализующие протоколы, основанные на TCP, являются потомками класса `TIdTCPServer` и используют введенные в нем механизмы обработки запросов. При этом каждый компонент предоставляет дополнительные средства, учитывающие специфику реализуемого протокола. В качестве примера рассмотрим компонент `IdDICTServer` (класс `TIdDICTServer`), реализующий TCP-протокол DICT, который позволяет создавать распределенные системы электронных словарей естественных языков. Клиенты DICT обращаются к серверу, связанному с базами данных словарей, с запросами, содержащими слова или термины, которые необходимо перевести. В ответ сервер посылает информацию, извлеченную из словаря. Протокол DICT поддерживает идентификацию и авторизацию клиентов, позволяет выбирать словарь из набора, доступного на сервере, и проводить поиск терминов при неполной информации, используя различные критерии и стратегии поиска. В рамках протокола DICT определен набор команд, посылаемых клиентом серверу, включающий такие команды как `Client` — идентификация клиента, `Auth` — авторизация клиента, `Define` — поиск слова в словаре и т. п.

Компонент `IdDICTServer` скрывает механизм обработки соединений класса `TIdTCPServer`, предоставляя в распоряжение программиста ряд событий, связанных с конкретными командами протокола DICT: `OnCommandAuth`, `OnCommandClient`, `OnCommandDefine`, `OnCommandHelp` и т. д.

Каждый из обработчиков этих событий получает ссылку на объект класса `TIdPeerThread`, который, как мы уже знаем, позволяет получить доступ к свойствам и методам объекта `TIdTCPConnection` для конкретного TCP-соединения. Кроме этого обработчик события может получать и другие аргументы, например параметры команды, вызвавшей событие.

Таким образом, написание сервера протокола DICT на основе компонента `IdDICTServer` фактически сводится к написанию обработчиков событий, связанных с командами протокола.

В процессе обработки команд мы должны возвращать клиенту лишь "содержательную часть" ответа сервера. Стандартные RFC-ответы с соответствующими статус-кодами будут сгенерированы автоматически.

До сих пор речь шла только о серверах, работающих с протоколами, основанными на TCP. Но Internet Direct позволяет работать и с другими протоколами, в частности со вторым широко применяемым в Интернете протоколом транспортного уровня — протоколом UDP. Протокол UDP относится к протоколам, не устанавливающим виртуальный канал между клиентом и сервером (и этим он отличается от протокола TCP). Вследствие этой своей особенности протокол UDP не позволяет отслеживать состояние передачи

отдельных пакетов и текущее состояние соединения. В частности в рамках протокола UDP не существует гарантированного метода определения разрыва соединения.

В связи со всем вышесказанным у читателя может сложиться впечатление, что протокол UDP "менее надежен", чем протокол TCP и в каком-то смысле это действительно так. Зачем же тогда нужен протокол UDP? Преимуществом протокола UDP является его сравнительная простота. По сравнению с TCP протокол UDP создает существенно меньшую нагрузку на сетевую систему, как на уровне приложений, так и на физическом уровне. При прочих равных условиях UDP-сервер способен обрабатывать одновременно больше запросов, чем TCP-сервер и обработка запросов выполняется быстрее. По этой причине протокол UDP используется в качестве основы для нескольких протоколов высокого уровня, в которых либо быстродействие является критическим фактором, либо доставка *всех* отправленных пакетов не обязательна. К первой категории относятся такие протоколы, как DNS (Domain Name System, протокол доменной (региональной) системы имен) и SNMP (Simple Network Management Protocol, простой протокол сетевого управления), ко второй — различные мультимедиа-протоколы, например IP-телефония.

Следует отметить, что для некоторых высокоуровневых протоколов, не предъявляющих жестких требований к транспортному слою (Chargen, Echo), набор компонентов Indy предоставляет как TCP, так и UDP варианты клиентов и серверов (разумеется, для того чтобы иметь возможность взаимодействовать между собой, клиент и сервер должны использовать один и тот же транспортный протокол).

Базовый Indy-сервер протокола UDP реализован в классе `TIdUDPServer`. К свойствам и методам своего предка — класса `TIdUDPBase` класс `TIdUDPServer` добавляет свойство `Bindings`, позволяющее связать сервер с определенными адресами и портами, свойство-событие `OnUDPRead` и некоторые другие элементы.

Поскольку в случае UDP-протокола сохранять информацию о состоянии соединения не нужно, сервер `IdUDPServer` не использует многопоточность. Обработка запросов UDP-сервером основана на других принципах. Рассмотрим тип процедуры-обработчика события `OnUDPRead`.

```
TUDPReadEvent = procedure (Sender: TObject; AData: TStream;
ABinding: TIdSocketHandle) of object;
```

Поток `AData` содержит данные, поступившие от клиента, а объект `ABinding`, кроме прочего, предоставляет информацию о локальных и удаленных адресе и порте.

Замечание

Напомним, что сервер может прослушивать несколько локальных адресов и/или портов, и нам, возможно, потребуется узнать, к какому именно адресу/порту обратился клиент.

Для того чтобы отправить клиенту ответ, воспользуемся методами `Send` или `SendBuffer`. Поскольку в рамках модели UDP постоянное соединение не устанавливается, методам отправки данных всегда следует явным образом указывать адрес и порт назначения. Получить эту информацию можно из свойств `PeerIP` и `PeerPort` объекта `ABinding`.

Напишем две программы, клиент и сервер, использующие протокол UDP для передачи файлов.

Клиент посылает серверу запрос в форме:

```
SEND имя_файла
```

где `имя_файла` — имя файла в файловой системе сервера.

В ответ на этот запрос сервер возвращает содержимое файла, либо сообщение об ошибке, если запрашиваемый файл не найден.

Исходные тексты программы сервера (C++) приводятся в листингах 5.5 и 5.6.

Листинг 5.5. UDP-сервер (заголовочный файл)

```
//---------------------------------------------------------------------------
#ifndef MainH
#define MainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <IdBaseComponent.hpp>
#include <IdComponent.hpp>
#include <IdUDPBase.hpp>
#include <IdUDPServer.hpp>
//---------------------------------------------------------------------------

class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TIdUDPServer *IdUDPServer1;
```

```
        void __fastcall IdUDPServer1UDPRead(TObject *Sender,
        TStream *AData, TIdSocketHandle *ABinding);
private:       // User declarations
public:        // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Листинг 5.6. UDP-сервер (основной файл)

```
//---------------------------------------------------------------------

#include <clx.h>
#include <string.h>
#pragma hdrstop

#include "Main.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------

void __fastcall TForm1::IdUDPServer1UDPRead(TObject *Sender,
      TStream *AData, TIdSocketHandle *ABinding)
{
  char cmd[512];
  int len = AData->Read(cmd, 512);
  cmd[4] = 0;
  if (strcmp(cmd, "SEND") == 0)
  {
```

```
    char * FN = &cmd[5];
    String FileName = FN;
    try
    {
      TFileStream * FS = new TFileStream(FileName, fmOpenRead);
    }
    catch (Exception * e)
    {
      IdUDPServer1->Send(ABinding->PeerIP, ABinding->PeerPort,
      "Ошибка при работе с файлом");
      if (FS != NULL) delete FS;
      return;
    }
    char * buf = (char*) malloc(FS->Size);
    FS->Read(buf, FS->Size);
    IdUDPServer1->SendBuffer(ABinding->PeerIP, ABinding->PeerPort,
    buf, FS->Size);
    free((void*)buf);
    delete FS;
  }
  else
  IdUDPServer1->Send(ABinding->PeerIP, ABinding->PeerPort,
  "Неверная команда");
}
//---------------------------------------------------------------
```

Текст сервера настолько прост, что вряд ли нуждается в дополнительных комментариях. Перед запуском сервера не забудем сделать две вещи: назначить хотя бы одну пару адрес/порт свойству `Bindings` и присвоить значение `true` свойству `Active` компонента `IdUDPServer1`.

Исходный текст UDP-клиента приводится в листингах 5.7 и 5.8.

Листинг 5.7. UDP-клиент (заголовочный файл)

```
//---------------------------------------------------------------------

#ifndef MainH
#define MainH
//---------------------------------------------------------------------
#include <Classes.hpp>
```

```
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <IdBaseComponent.hpp>
#include <IdComponent.hpp>
#include <IdUDPBase.hpp>
#include <IdUDPClient.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TIdUDPClient *IdUDPClient1;
        TEdit *FileName;
        TButton *LoadButton;
        TMemo *FileMemo;
        void __fastcall LoadButtonClick(TObject *Sender);
private:      // User declarations
public:       // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Листинг 5.8. UDP-клиент (основной файл)

```
//---------------------------------------------------------------------------

#include <clx.h>
#pragma hdrstop

#include "Main.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
```

```
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------

void __fastcall TForm1::LoadButtonClick(TObject *Sender)
{
   char buf[0x1000];
   int len;
   IdUDPClient1->Send("OPEN "+(String)FileName->Text+'\0');
   String S;
   while ((len = IdUDPClient1->ReceiveBuffer(buf, 0xFFF, 1000))!=0)
   {
     buf[len] = 0;
     S.printf("%s", buf);
   }
   FileMemo->Text = S;
}
//---------------------------------------------------------------------------
```

Кнопка `LoadButton` посылает запрос на загрузку файла, используя имя файла, введенное в компоненте `FileName`. Содержимое файла отображается в окне компонента `FileMemo`. Разумеется, наши простейшие клиент и сервер предназначены лишь для передачи сравнительно коротких текстовых файлов.

Если мы хотим организовать передачу файлов по сети с помощью протокола UDP, лучше воспользоваться Indy-сервером `TIdTrivialFTPServer`, реализующим протокол Trivial FTP (TFTP). Протокол TFTP можно рассматривать как упрощенный вариант протокола FTP (File Transfer Protocol, протокол передачи файлов). Этот протокол не поддерживает идентификацию и авторизацию клиентов, чтение и создание каталогов на удаленной машине, но позволяет организовать достаточно надежную передачу файлов в обоих направлениях.

Разместим в форме приложения компонент `IdTrivialFTPServer`. Как и для других компонентов-серверов, для этого компонента необходимо заполнить свойство `Bindings`. По умолчанию протокол TFTP использует порт 69, но если в нашей системе этот порт недоступен, назначим серверу другой порт. В главной форме нашего приложения мы разместим также компонент `TMemo`, который будет выполнять функции журнала.

В ответ на обращение клиента с запросом о передаче или приеме файла генерируется событие `OnReadFile` или `OnWriteFile`. По окончании операции вызывается обработчик события `OnTransferComplete`. Кроме этих событий,

класс `IdTrivialFTPServer` позволяет получить доступ ко всем событиям своего предка — класса `TIdUDPServer`.

В листинге 5.9 приводится текст обработчиков этих событий.

Листинг 5.9. Обработчики событий сервера TFTP

```
procedure TForm1.IdTrivialFTPServer1ReadFile(Sender: TObject;
  var FileName: String; const PeerInfo: TPeerInfo;
  var GrantAccess: Boolean; var AStream: TStream;
  var FreeStreamOnComplete: Boolean);
var
  S : String;
begin
  AStream := TFileStream.Create(FileName, fmOpenRead);
  S := Format('%s:%d connected for reading %s', [PeerInfo.PeerIP,
  PeerInfo.PeerPort, FileName]);
  Memo1.Lines.Add(S);
end;

procedure TForm1.IdTrivialFTPServer1WriteFile(Sender: TObject;
  var FileName: String; const PeerInfo: TPeerInfo;
  var GrantAccess: Boolean; var AStream: TStream;
  var FreeStreamOnComplete: Boolean);
var
  S : String;
begin
  AStream := TFileStream.Create(FileName, fmCreate);
  S := Format('%s:%d connected for writing %s', [PeerInfo.PeerIP,
  PeerInfo.PeerPort, FileName]);
  Memo1.Lines.Add(S);
end;

procedure TForm1.IdTrivialFTPServer1TransferComplete(Sender: TObject;
  const Success: Boolean; const PeerInfo: TPeerInfo; AStream: TStream;
  const WriteOperation: Boolean);
var
  S, Op, Res : String;
```

```
begin
  if WriteOperation then Op := 'Writing'
  else Op := 'Reading';
  if Success then Res := 'completed'
  else Res := 'faled';
  S := Format('%s:%d — %s operation %s', [PeerInfo.PeerIP,
  PeerInfo.PeerPort, Op, Res]);
  Memo1.Lines.Add(S);
end;
```

Как мы видим, в обработчиках приема/передачи файлов от нас требуется лишь создать объект класса `TFileStream`. Всю остальную работу по приему/передаче файла компонент `IdTrivialFTPServer` берет на себя. Параметр-переменная `FreeStreamOnComplete` позволяет указать, хотим ли мы, чтобы по завершении операции с файлом объект-поток был удален или нет. По умолчанию этот параметр имеет значение `true`.

Замечание

Обратим внимание на то, что источником/приемником передаваемых данных не обязательно должен быть файл, т. е. поток `TFileStream`. В принципе можно использовать для работы с данными и другие объекты-потоки.

Получить информацию об IP-адресе и порте клиента можно при помощи параметра `PeerInfo`. На основе этих данных можно организовать простейшую систему идентификации клиентов, основанную на их адресах.

Как уже отмечалось выше, базовый класс `TIdUDPServer` не является многопоточным. Однопоточная модель не создает затруднений, если обработка каждого запроса занимает немного времени. Однако в случае с протоколом TFTP это не так. Передача одного файла большого размера может занимать довольно продолжительное время даже при передаче по локальной сети. Сервер TFTP не способен обрабатывать запросы параллельно, но может накапливать их в очереди.

Напишем простую программу-клиент, принимающую и передающую файлы на наш сервер. Исходный текст приложения-клиента приводится в листинге 5.10.

Листинг 5.10. Клиент TFTP

```
unit Main;

interface

uses
  SysUtils, Types, Classes, Variants, QTypes, QGraphics, QControls,
```

```
  QForms, QDialogs, QStdCtrls, IdBaseComponent, IdComponent, IdUDPBase,
  IdUDPClient, IdTrivialFTP;

type
  TForm1 = class(TForm)
    IdTrivialFTP1: TIdTrivialFTP;
    PutButton: TButton;
    GetButton: TButton;
    OpenDialog1: TOpenDialog;
    FileName: TEdit;
    procedure PutButtonClick(Sender: TObject);
    procedure GetButtonClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.PutButtonClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    if FileExists(OpenDialog1.FileName) then
    begin
      IdTrivialFTP1.Put('Money.mp3', 'Money.mp3');
      IdTrivialFTP1.Put(OpenDialog1.FileName,
      ExtractFileName(OpenDialog1.FileName));
    end;
  end;
end;
```

```
procedure TForm1.GetButtonClick(Sender: TObject);
var
  FS : TFileStream;
begin
  IdTrivialFTP1.Get(FileName.Text,
  ExtractFileName(OpenDialog1.FileName));
end;
end.
```

Отправка файла осуществляется компонентом-кнопкой `PutButton`, а загрузка — компонентом-кнопкой `GetButton`. При отправке данных на сервер для выбора файла используется диалоговое окно компонента `OpenDialog1`, а при приеме данных — значение элемента ввода `FileName`.

Компоненты-клиенты Indy

В предыдущих разделах этой главы мы уже пользовались различными компонентами-клиентами Indy для демонстрации работы соответствующих серверов. Кроме того, в предыдущей главе *(см. раздел "Программирование сетевых демонов" гл. 4)*, мы рассмотрели компонент-клиент протокола HTTP — `IdHTTP`.

Все компоненты-клиенты, инкапсулирующие TCP-протоколы, основаны на классе `TIdTCPConnection`, тогда как предком компонентов-клиентов, инкапсулирующих протоколы, основанные на UDP является класс `TIdUDPBase`.

Каждый экземпляр компонента-клиента TCP инкапсулирует одно TCP-соединение, так что если мы захотим организовать параллельную загрузку нескольких ресурсов с TCP-сервера (как это делают современные Web-браузеры и програмы-качалки), нам понадобится несколько экземпляров соответствующих объектов. Важнейшими свойствами, наличествующими у всех TCP-клиентов, являются `Host` и `Port`, указывающие соответственно имя хоста (или IP-адрес) и порт сервера, с которым устанавливается соединение. Для установки соединения служит метод `Connect`, в случае успеха вызывающий событие `OnConnected`.

Компоненты-клиенты Indy могут служить основой полноценных интернет-приложений. В доказательство этого рассмотрим программу FTP-клиент, основанную на компоненте `IdFTP`.

На рис. 5.6 показана главная форма приложения, с размещенными в ней компонентами.

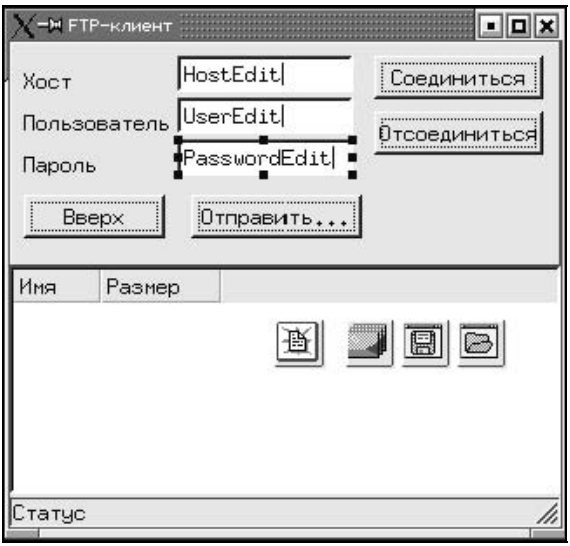


Рис. 5.6. Форма приложения FTP-клиента

Для отображения содержимого FTP-каталогов мы используем компонент `ListView` с двумя колонками: `Имя` и `Размер`, отображающими соответственно имена и размеры файлов. Для того чтобы сделать содержимое каталога более наглядным, в приложение добавлен компонент `ImageList`, включающий пиктограммы для элементов списка `ListView` (файлы и каталоги). На рис. 5.7 показано окно редактора компонента `ImageList` с пиктограммами. Элемент 0 — пиктограмма для файлов, элемент 1 — пиктограмма для каталогов.

Компоненты `OpenDialog` и `SaveDialog` служат для выбора имени файла при приеме и отправке.

Компонент `IdFTP` позволяет получить информацию о состоянии соединения при помощи события `OnStatus`. Мы используем компонент `StatusBar` для вывода текстовой информации, предоставляемой обработчику `OnStatus`.

Для установки связи с FTP-сервером нам требуется некоторая информация: имя узла (мы используем стандартные порты FTP), имя пользователя и пароль. Для этого используются строки ввода `HostEdit`, `UserEdit` и `PasswordEdit`.

При щелчке левой кнопкой мыши для отображения каталога в окне `ListView` выполняется переход в этот каталог (FTP-команда `chdir`), при щелчке по имени файла начинается загрузка файла на локальный компьютер. При этом открывается диалоговое окно, в котором можно выбрать имя файла в локальной файловой системе. Для перехода в каталог верхнего уровня служит кнопка **Вверх**, а для передачи файла на удаленный сервер используется кнопка **Отправить** (при этом также открывается диалоговое окно, позволяющее выбрать файл в локальной системе).

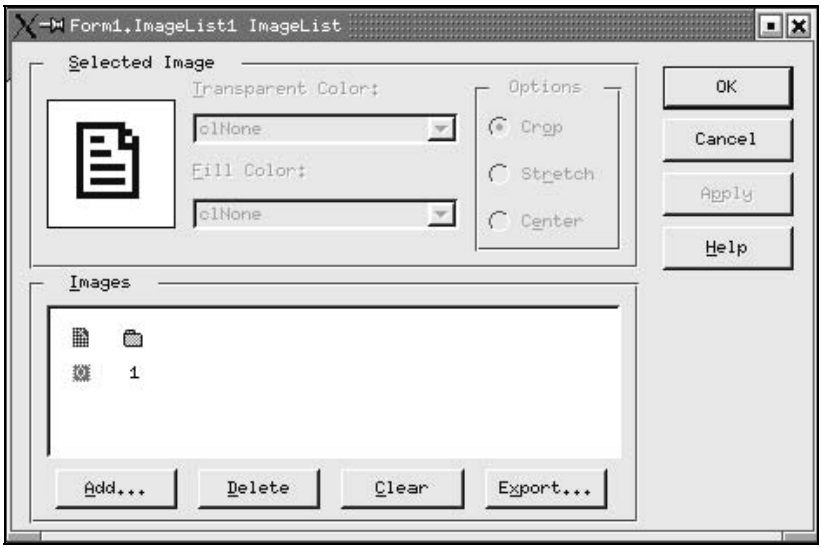


Рис. 5.7. Редактор компонента `ImageList`

Исходный текст FTP-клиента приводится в листинге 5.11.

Листинг 5.11. FTP-клиент

```
unit Main;

interface

uses
  SysUtils, Types, Classes, Variants, QTypes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls, QImgList, QComCtrls, IdBaseComponent,
  IdComponent, IdTCPConnection, IdTCPClient, IdFTP, IdFTPList, QExtCtrls;

type
  TForm1 = class(TForm)
    ListView1: TListView;
    ImageList1: TImageList;
    Panel1: TPanel;
    ConnectButton: TButton; // Кнопка "Соединиться"
    StoreButton: TButton;   // Кнопка "Отправить"
    DisconnectButton: TButton; // Кнопка "Отсоединиться"
    HostEdit: TEdit;      // Поле "Хост"
```

```
    UserEdit: TEdit;      // Поле "Пользователь"
    PasswordEdit: TEdit;  // Поле "Пароль"
    IdFTP1: TIdFTP;
    UpDirButton: TButton; // Кнопка "Вверх"
    SaveDialog1: TSaveDialog;
    OpenDialog1: TOpenDialog;
    StatusBar1: TStatusBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure ConnectButtonClick(Sender: TObject);
    procedure DisconnectButtonClick(Sender: TObject);
    procedure ListView1ItemClick(Sender: TObject; Button: TMouseButton;
      Item: TListItem; const Pt: TPoint; ColIndex: Integer);
    procedure UpDirButtonClick(Sender: TObject);
    procedure StoreButtonClick(Sender: TObject);
    procedure IdFTP1Status(ASender: TObject; const AStatus: TIdStatus;
      const AStatusText: String);
    procedure ListView1KeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure FillDir;
    procedure ProcessItem(Item : TListItem);
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}


procedure TForm1.FillDir;
var
```

```
  i, dir_ind : Integer;
  NewItem : TListItem;
begin
  ListView1.Items.Clear;
  dir_ind := 0;
  for i := 0 to IdFTP1.DirectoryListing.Count – 1 do
  begin
    if IdFTP1.DirectoryListing.Items[i].ItemType = ditDirectory then
    begin
      if ListView1.Items.Count = dir_ind then
      NewItem := ListView1.Items.Add
      else NewItem := ListView1.Items.Insert(dir_ind);
      Inc(dir_ind);
      NewItem.ImageIndex := 1
    end else
    begin
      NewItem := ListView1.Items.Add;
      NewItem.ImageIndex := 0;
    end;
    NewItem.Caption := IdFTP1.DirectoryListing.Items[i].FileName;
    NewItem.SubItems.Add(
    IntToStr(IdFTP1.DirectoryListing.Items[i].Size));
  end;
end;

procedure TForm1.ProcessItem(Item : TListItem);
var
  S : String;
begin
  S := IdFTP1.RetrieveCurrentDir+'/'+Item.Caption;
  if Item.ImageIndex = 1 then
  begin
    IdFTP1.ChangeDir(S);
    IdFTP1.List(nil);
    FillDir();
  end else
  begin
    SaveDialog1.FileName := ExtractFileName(S);
```

```
    if SaveDialog1.Execute() then
    IdFTP1.Get(S, SaveDialog1.FileName);
  end;
end;

procedure TForm1.ConnectButtonClick(Sender: TObject);
begin
  IdFTP1.Host := HostEdit.Text;
  IdFTP1.Username := UserEdit.Text;
  IdFTP1.Password := PasswordEdit.Text;
  IdFTP1.Connect();
  IdFTP1.List(nil);
  FillDir();
end;

procedure TForm1.DisconnectButtonClick(Sender: TObject);
begin
  IdFTP1.Disconnect();
end;

procedure TForm1.ListView1ItemClick(Sender: TObject;
  Button: TMouseButton;
  Item: TListItem; const Pt: TPoint; ColIndex: Integer);
begin
  if Button = mbLeft then ProcessItem(Item);
end;

procedure TForm1.UpDirButtonClick(Sender: TObject);
begin
  IdFTP1.ChangeDirUp();
  IdFTP1.List(nil);
  FillDir();
end;

procedure TForm1.StoreButtonClick(Sender: TObject);
begin
  if OpenDialog1.Execute() then
```

```
  begin
    IdFTP1.Put(OpenDialog1.FileName,
    ExtractFileName(OpenDialog1.FileName));
    IdFTP1.List(nil);
    FillDir();
  end;
end;

procedure TForm1.IdFTP1Status(ASender: TObject; const AStatus: TIdStatus;
  const AStatusText: String);
begin
  StatusBar1.Panels[0].Text := AStatusText;
end;

procedure TForm1.ListView1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
    4100 : // Enter
      ProcessItem(ListView1.Selected);
    4099 : // BackSpace
      begin
        IdFTP1.ChangeDirUp();
        IdFTP1.List(nil);
        FillDir();
      end;
  end;
end;

end.
```

Рассмотрим сначала две вспомогательные процедуры `FillDir` и `ProcessItem`. Процедура `FillDir` заполняет компонент `ListView` данными, полученными при чтении содержимого удаленного каталога (FTP-команда `list`). Получить список элементов каталога можно при помощи метода `List`. Метод `List` позволяет получить данные о текущем каталоге. После соединения и входа на сервер текущий каталог определяется сервером. Получить полный путь к текущему каталогу в виртуальной файловой системе

FTP-сервера можно при помощи метода RetrieveCurrentDir. Метод List может вернуть список элементов каталога в объекте TStrings (если ему передана ссылка на соответствующий объект), однако формат данных, полученных таким способом не очень удобен для анализа, и мы воспользуемся другим приемом. Каждый раз после вызова метода List компонент IdFTP заполняет данными свойство DirectoryListing. Свойство DirectoryListing представляет собой коллекцию элементов типа TIdFTPListItem, каждый из которых хранит информацию об одном из элементов каталога.

В процедуре FillDir мы сначала очищаем список ListView1, а затем перебираем все элементы списка IdFTP1DirectoryListing, заполняя при этом список ListView1 новыми элементами. Заполнение компонента ListView1 осуществляется таким образом, чтобы элементы каталога, являющиеся каталогами, следовали в начале списка.

Процедура ProcessItem выполняет операции над элементами списка ListView1. Если переданный процедуре элемент представляет собой отображение каталога, осуществляется переход в новый каталог и чтение его содержимого. Смена каталога выполнняется при помощи метода ChangeDir. При вызове метода ChangeDir следует указывать полный путь к целевому каталогу, поэтому мы используем метод RetrieveCurrentDir.

Если переданный процедуре ProcessItem элемент представляет собой отображение файла, начинается процесс загрузки файла. Прежде всего вызывается диалоговое окно, позволяющее пользователю выбрать имя и расположение загружаемого файла в локальной файловой системе. Если пользователь согласен продолжить загрузку, вызывается метод Get. В классе TIdFTP существует два перегруженных варианта метода Get. Один вариант позволяет записывать получаемые с сервера данные в поток TStream. Другой вариант (которым мы и пользуемся) позволяет указать имя файла для сохранения данных на диске.

Установка связи с сервером осуществляется при помощи метода Connect. Обычно соединение с сервером выполняется в два этапа: сначала устанавливается TCP-соединение, затем производится идентификация и авторизация пользователя. Метод Connect позволяет объединить оба этапа. Для этого параметру AAutoLogin следует присвоить значение true (что и сделано по умолчанию). Данные об узле метод Connect получает из свойств Host и Port класса TIdFTP, а реквизиты пользователя — из свойств UserName и Password.

При работе с FTP-клиентом следует учесть одну особенность. В силу специфики протокола FTP (необходимость хранить информацию о каждом подключенном пользователе, интенсивность трафика при передаче файлов)

FTP-сервер обычно ограничивает число одновременных подключений и разрывает соединение при длительном отсутствии активности. Что понимать под "длительным отсутствием активности", зависит от сервера (точнее, от его администратора), и мы должны учитывать, что в отсутствие передачи команд или файлов сервер может разорвать соединение в любой момент. Для того чтобы проинформировать пользователя нашей программы о таком разрыве, необходимо назначить обработчик свойству `OnStatus` или свойству `OnDisconnected` компонента `IdFTP`.

Проверить работу нашего FTP-клиента можно и на локальной машине. Для этого следует убедиться сначала в том, что в локальной системе запущен FTP-сервер (обычно это демон `proftpd`). Если сервер запущен, наберем в строке **Хост** значение **127.0.0.1**, в строке **Пользователь** — свое имя пользователя, а в строке **Пароль** — свой пароль для входа в систему. При стандартном уровне безопасности ОС Linux каждый зарегистрированный пользователь системы имеет учетную запись FTP, позволяющую ему управлять своими файлами и каталогами с другого компьютера.

Вообще-то такой способ удаленного управления нельзя признать безопасным, так как реквизиты пользователя для входа в систему передаются по сети, и злоумышленник, перехвативший эти реквизиты, может получить полный контроль над нашими файлами. По этой причине пользователь root обычно не имеет учетной записи FTP.

Работающий FTP-клиент показан на рис. 5.8.

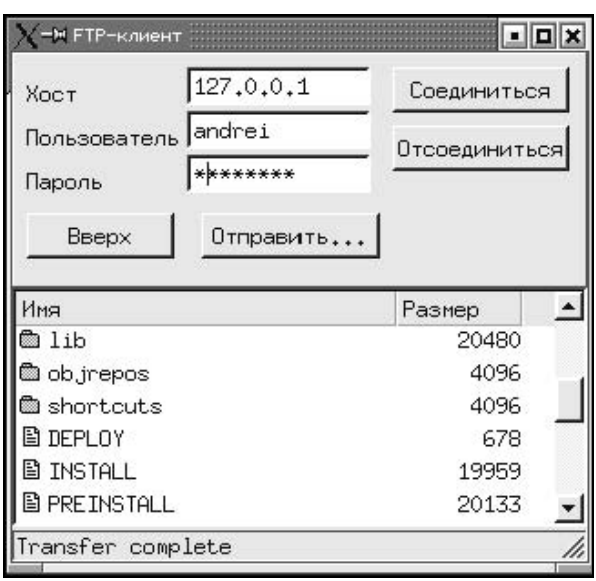


Рис. 5.8. Клиент FTP

Компоненты-перехватчики Indy и протокол SSL

Следует обратить внимание на свойство `Intercept`, введенное в классе `TIdTCPConnection` и доступное во всех его потомках. Это свойство типа `TIdConnectionIntercept` позволяет устанавливать перехватчики Indy. Объекты-перехватчики дают возможность контроль над соединением на более низком уровне по сравнению с компонентом `TIdTCPConnection`. С помощью объектов-перехватчиков возможно управлять состоянием соединения и операциями ввода/вывода, а также модифицировать потоки данных таким образом, что все выполняемые операции будут "прозрачны" с точки зрения соответствующего компонента. Классическим примером применения компонентов-перехватчиков может служить шифрование данных. Компонент-перехватчик может выполнять шифрование исходящих и дешифрование входящих данных в рамках традиционных протоколов, при этом взаимодействие с компонентами Indy на более высоком уровне не претерпит никаких изменений.

Практическим примером компонентов-перехватчиков может служить класс `TIdConnectionInterceptOpenSSL`, позволяющий организовать обмен данными между клиентами и серверами при помощи протокола SSL (Secure Sockets Layer, "слой защищенных соединений") — протокола безопасного обмена данными между Web-приложениями. Протокол SSL является "прослойкой" между уровнем приложения и сетевым уровнем. SSL использует методы шифрования, гарантирующие конфиденциальность и аутентичность данных, передаваемых в сети.

В силу юридических ограничений в состав Indy не входят библиотеки, реализующие протокол SSL. Если есть право на использование SSL, мы должны загрузить и установить необходимые компоненты самостоятельно.

Чтобы использовать протокол SSL на сервере, необходимо получить специальный сертификат, подтверждающий аутентичность сервера. Сертификат позволяет производить электронные операции с денежными средствами, предоставлять закрытую информацию в Интернет, предоставлять доступ с удаленных серверов, подтверждать электронную подпись и т. п. Более подробные сведения о правомочности использования SSL и связанных с этим правилах можно найти в текущем законодательстве.

Другой пример использования перехватчиков — класс `TIdLogDebug`. Этот класс позволяет сохранять на диске полную информацию о принимаемых и передаваемых данных, а также об изменениях состояния соединения. Как следует из названия самого класса, он предназначен в основном для отладки приложений.

Другие компоненты Indy

В предыдущих разделах этой главы мы познакомились с компонентами-клиентами и компонентами-серверами Indy, а также с некоторыми вспомогательными компонентами. В этом разделе мы кратко опишем некоторые дополнительные компоненты, расположенные на странице **Indy Misc** палитры компонентов.

Компонент *IdCookieManager*

Компонент `IdCookieManager` позволяет HTTP-клиенту организовать обработку cookies — небольших блоков информации, которые Web-сервер посылает клиенту для сохранения каких-либо промежуточных данных (напомним, что HTTP-сервер не сохраняет информации об удаленных пользователях в перерывах между транзакциями, метод cookies позволяет возложить эту задачу на HTTP-клиента).

Компонент `IdCookieManager` поддерживает список полученных от сервера записей в свойстве `CookieCollection`. Элементами этого списка являются объекты, хранящие информацию в соответствии с требованиями RFC 2109 и RFC 2965.

При добавлении нового элемента в список `CookieCollection` вызывается событие `OnNewCookie`, которое позволяет клиенту принять или отвергнуть новую запись, исходя из переданной в ней информации. При помощи методов `AddCookie` и `AddCookie2` клиент может самостоятельно добавлять записи в список `CookieCollection`, например, из файлов, хранящихся на дисках.

Остальные операции с cookies выполняется клиентом автоматически.

Компонент *IdIPWatch*

Компонент `IdIPWatch`. позволяет определить текущее состояние сетевой подсистемы ОС Linux, а также предыдущие состояния. Свойство `CurrentIP` возвращает последний использованный IP-адрес (в одной системе может быть несколько адресов). Свойство `IPHistoryList` типа `TStringList` содержит список использовавшихся IP-адресов. Сохранить этот список можно в файле, имя которого задано в свойстве `HistoryFilename` (для того чтобы сохранить список в файле, необходимо вызвать метод `SaveHistory`).

Свойство `IsOnline` позволяет проверить, подключен ли компьютер к сети. Это свойство возвращает значение `true`, если значение `CurrentIP` непустое и отличается от **127.0.0.1**. Частоту, с которой проверяется состояние локальной сетевой системы, можно указать при помощи свойства `WatchInterval`. Если нужно проверить состояние системы в данный момент, следует вызвать метод `ForceCheck`. При изменении состояния подключения к сети вызывается событие `OnStatusChanged`.

Компоненты-кодировщики

Многие популярные Интернет-протоколы, например почтовые, изначально создавались исключительно для передачи текстовой информации. В англоязычном мире это означало, что сообщение могло состоять только из первых 128 символов 256 символьного набора, которые можно было кодировать семью, а не восемью битами. По этой причине семибитные кодировки стали весьма распространены. Однако возможности сетей развивались, и в скором времени возникла потребность передавать по сети двоичные данные (графические и программные файлы или текст, написанный не латинским шрифтом). Еще одной ситуацией, в которой семибитные кодировки оказались неприемлемы, стало шифрование передаваемой по сети информации. Поскольку заменить используемые протоколы представлялось затруднительным, возникла проблема передачи восьмибитных значений в семибитных кодировках.

Решением этой проблемы являются технологии, которые представляют двоичные восьмибитные данные в семибитном виде.

В Internet Direct указанные задачи решаются набором компонентов, основанных на базовых классах `TIdEncoder` и `TIdDecoder`.

У класса `TIdEncoder` три метода: два перегруженных метода `Encode` и метод `EncodeString`. Все три метода выполняют сходные операции: получают данные (в различных форматах) и возвращают строку, содержащую закодированные данные.

Класс `TIdDecoder`, служащий основой для компонентов, выполняющих декодирование, также включает три метода: `DecodeString`, `DecodeToStream` и `DecodeToString`. Все три метода получают закодированные данные в форме строки. Методы `DecodeString` и `DecodeToString` возвращают декодированные данные также в переменной типа `String`, а метод `DecodeToStream` записывает их в поток, представленный ссылкой на класс `TStream`.

Среди наследников классов `TIdEncoder`/`TIdDecoder` следует упомянуть классы `TIdEncoderUUE` и `TIdDecoderUUE`, реализующие кодирование/декодирование данных в соответствии с алгоритмом UUEncode.

Классы `TIdEncoderMIME` и `TIdDecoderMIME` выполняют преобразование данных в соответствии с популярной в почтовых протоколах кодировкой MIME. Для работы с кодировкой Quoted Printable предназначены специальные классы `TIdEncoderQuotedPrintable` и `TIdDecoderQuotedPrintable`.

Компонент *IdDateTimeStamp*

Этот компонент позволяет работать с форматами даты и времени, принятыми в различных интернет-протоколах. Возможности класса `TIdDateTimeStamp` не ограничиваются преобразованием даты и времени из одного интернет-формата в другой. Этот класс позволяет также анализировать значения даты и времени и представлять их в удобной для пользователя форме.

Назначить классу `TIdDateTimeStamp` значение даты и времени для дальнейших преобразований можно несколькими способами, и выбор конкретного способа зависит от исходного формата значения даты и времени. Метод `SetFromTDateTime` позволяет задать значение даты и времени в традиционном для Borland формате `TDateTime`, а метод `SetFromTTimeStamp` присваивает компоненту значение в формате `TTimeStamp`, принятом в файловой системе.

Метод `SetFromISO8601` позволяет задать значение даты и времени в формате, принятом в стандарте ISO 8601. Данному методу передается строка, содержащая описание даты и времени в одном из трех форматов, допускаемых стандартом ISO 8601:

- YYYY-MM-DD[T] Hr:Min:Sec;
- YYYY-NNN[T] Hr:Min:Sec;
- YYYY-WNN-D[T] Hr:Min:Sec.

В этих форматах YYYY означает год (четырехзначное число), MM — порядковый номер месяца (от 01 до 12), NNN — день года (от 001 до 365) WNN — номер недели (буква "W" является частью формата), а следующее за номером недели число D — номер дня недели. За указанием даты следует указание времени, которое может быть отделено от даты символом "T" или пробелом. Таким образом, полдень 1 апреля 2002 года может быть представлен в форматах ISO 8601 как:

- `2002-04-01 12:00:00`;
- `2002-091T12:00:00`;
- `2002-W14-2 12:00:00` (не забудем, что английская неделя начинается с воскресенья).

Метод `SetFromRFC822` позволяет задать значение даты и времени в формате, принятом в стандарте RFC 822. Для 12 часов первого апреля 2002 года строка в этом формате будет выглядеть как `Mon, 1 Apr 2002 12:00:00`.

Назначив компоненту `IdDateTimeStamp` значение даты и времени в одном из вышеперечисленных форматов, можно затем считать это значение в любом из этих форматов. Для этого можно воспользоваться методами, имена которых начинаются с префикса "Get", например `GetAsRFC822`, или предназначенными только для чтения свойствами с именами, начинающимися с префикса "As", например `AsTTimeStamp`.

Ряд свойств позволяют использовать компонент `IdDateTimeStamp` в качестве календаря:

- `DayOfWeek` — возвращает число, соответствующее номеру дня недели (как уже было отмечено, первый день — воскресенье);
- `DayOfWeekName` — возвращает английское название дня недели;

- `DaysInYear` — возвращает количество дней в указанном году;
- `IsLeapYear` — позволяет определить, является ли указанный год високосным.

Некоторые свойства компонента предназначены для форматированного вывода.

- `HourOf12Day` — возвращает значение часа в формате 1-12.
- `HourOf24Day` — возвращает значение часа в формате 00-23.
- `IsMorning` — позволяет определить, является ли указанный час дополуденным или послеполуденным. Это свойство удобно использовать совместно с `HourOf12Day`.

Компонент `IdDateTimeStamp` позволяет модифицировать значение даты и времени при помощи методов, увеличивающих или уменьшающих значения отдельных компонентов даты/времени. Методы, увеличивающие значение, начинаются с префикса "Add", а методы, уменьшающие значение — с префикса "Subtract". Например, метод `AddHours` позволяет увеличить значение часов, метод `AddTDateTime` — прибавить к заданному значению даты и времени значение в формате `TDateTime`, а метод `SubtractMilliseconds` — вычесть из заданного значения определенное число миллисекунд.

Глава 6

Язык XML и его производные — основа современных Web-технологий

Эту главу следует рассматривать как вводную к следующим трем главам, посвященным разработке приложений для Web-сервера. Технологии программирования Web-приложений в среде Kylix (да и сама среда разработки) настолько тесно срослись с технологиями XML, что понимание последних совершенно необходимо для умелого применения средств разработки Web-приложений.

Язык XML

XML (Extensible Markup Language, *расширяемый язык разметки*) — это язык разметки, описывающий класс объектов данных, называемых XML-документами. Языком разметки документов называется набор инструкций (их называют тэгами или управляющими дескрипторами), предназначенных для формирования в документах какой-либо структуры и определения отношений между различными элементами этой структуры. Тэги выделяются относительно основного содержимого документа специальными синтаксическими элементами и служат в качестве инструкций для программы, производящей отображение содержимого документа или выполняющей содержащиеся в документе инструкции. В современных языках разметки для выделения тэгов используются символы < и >, внутри которых помещаются имена тэгов и их параметров. Для обеспечения прозрачности языков разметки относительно содержательного текста в них вводятся специальные средства для отображения символов < и >, и других управляющих символов языка в тех случаях, когда эти символы не несут синтаксической нагрузки.

Термин *расширяемый* в названии языка XML означает, что XML является не столько готовым языком оформления документов, как, например, язык гипертекстовой разметки HTML, сколько *мета-языком*, позволяющим создавать описания грамматик других языков разметки и выполнять синтаксический анализ документов, созданных на этих языках.

В языке XML тэги не имеют определенного лексического значения, XML лишь определяет структуру тэгов и порядок их формирования. Благодаря этому язык XML очень хорошо подходит для описания произвольных иерархических структур. Рассмотрим простой пример. Допустим, мы хотим составить описание объекта "комната" (`Room`) с указанием иерархических отношений между элементами этого объекта. Объект `Room` содержит следующие элементы: внешние элементы (`Externals`), в число которых входят окно (`Window`) и дверь (`Door`), и внутренние элементы (`Internals`), включающие стол (`Table`) и кровать (`Bed`).

Иерархическая схема объекта `Room` представлена на рис. 6.1.

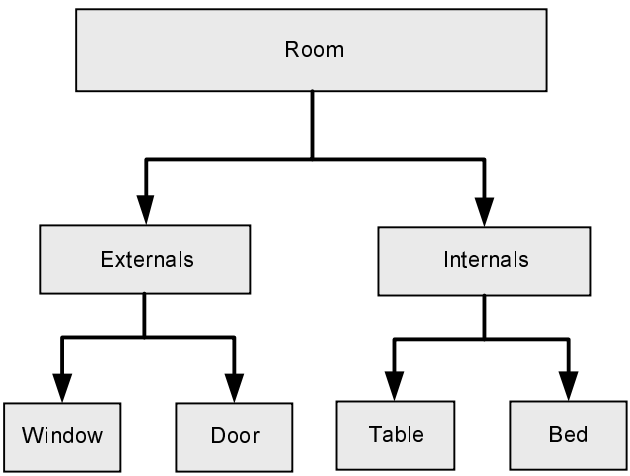


Рис. 6.1. Иерархия элементов объекта `Room`

На языке XML те же иерархические отношения могут быть выражены при помощи структуры, приведенной в листинге 6.1.

Листинг 6.1. Описание иерархии элементов объекта `Room` на языке XML

```
<Room>
<Externals>Window</Externals>
<Externals>Door</Externals>
<Internals>Table</Internals>
<Internals>Bed</Internals>
</Room>
```

Как видим, каждому элементу объекта `Room`, содержащему элементы нижнего иерархического уровня, в том числе самому объекту `Room`, можно присвоить соответствующий тэг. Таким образом, становится очевидным, что

язык XML лучше всего подходит для описания древовидных иерархических структур данных.

Еще одним из очевидных достоинств языка XML является возможность использования его в качестве универсального языка запросов к хранилищам информации. Именно язык XML лежит в основе протокола SOAP.

Язык XML позволяет также осуществлять контроль за корректностью данных, хранящихся в документах, проводить проверку иерархических соотношений внутри документа и устанавливать единый стандарт на структуру документов, содержащих определенные данные. Эта особенность языка XML означает, что его можно использовать при построении сложных информационных систем, в которых производится обмен информацией между различными приложениями, работающими, возможно, на разных платформах. Фактически язык XML позволяет создать стандартную структуру механизма обмена информацией в самом начале работы над проектом, которая станет основой для всех создаваемых в рамках проекта приложений.

Структура XML-документа

Структура простейшего XML-документа представлена в листинге 6.2.

Листинг 6.2. Простейший XML-документ

```
<?xml version="1.0"?>
<!-- Это комментарий --!>
<document_body>
Это тело документа. Выделяющий его тэг является произвольным.
</document_body>
```

Тело документа XML состоит из элементов разметки (markup) и содержимого документа (content). XML-тэги предназначены для определения элементов документа, их атрибутов и других конструкций языка. Данные, заключенные в угловые скобки < и > (имя тэга и его атрибуты) мы будем называть определением тэга.

Любой XML-документ должен всегда начинаться с конструкции `<?xml?>`, внутри которой можно задавать номер версии языка, номер кодовой страницы, а также некоторые другие параметры.

При создании XML-документов необходимо учитывать следующие правила, характерные для языка XML:

- документ должен начинаться с конструкции `<?xml?>`, в которой указывается язык разметки документа, номер его версии и дополнительная информация;

- для каждого открывающего тэга, определяющего некоторую область данных в документе, обязательно должен существовать комплиментарный закрывающий тэг (этим язык XML отличается от языка HTML, который позволяет пропускать закрывающие тэги);
- в языке XML учитывается регистр символов;
- все значения атрибутов, используемых в определении тэгов, должны быть заключены в кавычки;
- вложенность тэгов в языке XML строго контролируется, поэтому необходимо следить за порядком следования открывающих и закрывающих тэгов;
- вся информация, располагающаяся между начальным и конечным тэгами, рассматривается в языке XML как данные, и поэтому учитываются все символы форматирования (то есть пробелы, символы перевода строки, табуляции не игнорируются, как в языке HTML).

Содержимое XML-документа представляет собой набор элементов данных, секций `CDATA`, директив анализатора, комментариев, специальных символов и текста.

Элементы данных являются важнейшей структурной единицей XML-документа. Конструкция `<Internals>Table</Internals>` определяет непустой элемент данных `<Internals>`, содержимым которого является объект `Table`. В качестве содержимого элементов данных могут выступать любые элементы XML: простой текст, другие элементы данных, секции `CDATA`, инструкции по обработке, комментарии и т. п. Любой непустой элемент должен состоять из открывающего и закрывающего тэгов и данных, заключенных между ними. Некоторые элементы могут состоять из одного тэга и не иметь вложенных данных. Поскольку в языке XML любой открывающий тэг должен иметь соответствующий ему закрывающий тэг, такие "непарные" тэги должны меть форму: `<single_tag/>`.

Кроме имени, тэг может содержать один или несколько параметров, которые, как правило, описывают дополнительные характеристики соответствующего элемента данных. Примером тэга с параметром может служить тэг, определяющий толщину линии: `<line_width unit="mm">1.5</line_width>`. Обратим внимание на то, что значение параметра `unit` заключено в кавычки.

Выше уже говорилось о том, что язык XML должен быть прозрачен относительно содержащегося в документе текста. Это требование означает, что в рамках языка XML должны быть предусмотрены средства отображения в тексте символов, являющихся управляющими символами языка XML. Для решения этой проблемы в языке XML введены специальные формы представления управляющих символов языка в тексте. В табл. 6.1 приводятся некоторые управляющие символы языка XML и различные методы их текстового представления.

Таблица 6.1. *Текстовые представления управляющих символов языка XML*

Символ	Символьная форма	Десятичная форма	Шестнадцатеричная форма
<	&lt	&#060	&#x3C
>	&gt	&#062	&#x3E
&	&amp	&#038	&#x26

Например, фрагмент XML-документа `&lt&amp&gt` будет отображен браузером как `<&>`.

Директивы анализатора представляют собой специальные тэги, предназначенные не для отображения данных документа, а для передачи команд и данных программе, выполняющей разбор XML-документа. Тэги директив отличаются от остальных тэгов языка XML тем, что являются непарными и не имеют закрывающей формы. Общая структура тэга директивы имеет вид: `<?some directive?>`.

Примером директивы анализатора может служить конструкция `<?xml version="1.0"?>`, расположенная в начале XML-документа (листинг 6.2).

Комментарии языка XML заключаются в специальные тэги, которые пропускаются анализатором при разборе XML-документов. Таким образом, текст комментариев анализатор попросту игнорирует. Как и директивы анализатора, тэги комментариев являются непарными и не имеют закрывающей формы. Общая структура тэга комментария имеет вид: `<!-- ... --!>`. Пример комментария также можно найти в листинге 6.2.

Блоки `CDATA` позволяют задать область документа, внутри которой управляющие символы и инструкции языка XML не будут обрабатываться анализатором при разборе, но которая, в отличие от комментариев, не будет игнорироваться в приложении.

Для определения блока `CDATA` необходимо использовать тэги `<![CDATA] ... ]]>`. Разумеется, необходимо следить за тем, чтобы в области, ограниченной этими тэгами, не встречалась последовательность символов `]]`.

Среда Kylix использует XML-документы в целом ряде новых компонентов, предназначенных для работы с наборами данных: `XMLTransform`, `XMLTransformProvider` и `XMLTransformClient`. Компонент `XMLDocument` позволяет организовать в приложении синтаксический разбор структуры данных, описываемых в XML-документе. Кроме того, в наборе компонентов-генераторов контента появился компонент `XSLPageProducer`, позволяющий генерировать HTML-страницы на основе данных, представленных XML-документом и XSL-шаблоном.

Вместе с тем язык XML должен представлять интерес для программиста, работающего в среде Kylix, не только сам по себе, но и как основа для целого ряда языков, использующихся различными технологиями, поддержка которых реализована в Kylix. Именно этим языкам и посвящена оставшаяся часть данной главы.

Создание новых языков на основе XML. DTD-описания

Итак, язык XML позволяет создавать новые языки для описания и отображения данных. Для создания новых языков необходим некий формальный механизм описания их грамматики, т. е. перечня допустимых элементов данных, их возможного содержимого и атрибутов. Наиболее распространенным методом описания грамматики XML-языков на сегодняшний день являются описания в формате DTD (Documents Type Definitions, определения типа документа).

В XML-документах форматом DTD определяется набор допустимых элементов данных, перечень элементов данных, которые могут входить в состав других элементов данных, а также допустимые атрибуты для каждого элемента данных. Использование формата DTD — описаний для XML-документов, необходимо лишь в том случае, если требуется формальное описание синтаксиса документа (например, для проверки его синтаксическим анализатором). В принципе, документы, созданные без этих описаний, будут корректно обрабатываться программой-анализатором, если, конечно, они удовлетворяют основным требованиям синтаксиса языка XML. Использование DTD-описаний позволяет применять для автоматической генерации документов заданного синтаксиса специальные универсальные программы.

Обычно (хотя и необязательно) DTD-описания хранятся в отдельных файлах, а XML-документы содержат лишь ссылки на эти файлы-описания.

В DTD-описании для языка XML используются следующие типы правил:

- правила для элементов данных и их атрибутов;
- описания категорий (макроопределений);
- описание форматов бинарных данных.

Эти правила описывают основные конструкции языка: элементы данных, атрибуты, символьные константы и внешние файлы данных.

Напишем DTD-определение для XML-документа, описывающего объект `Room` (листинг 6.1). Поскольку у элементов данных этого документа нет атрибутов, DTD-определение для него будет состоять исключительно из описаний элементов данных.

Элемент данных в DTD-описании определяется с помощью дескриптора `!ELEMENT`, в котором указывается название элемента и структура его содержимого. Например, для элемента данных `Internals` описание будет выглядеть следующим образом:

```
<!ELEMENT Internals (#PCDATA)>
```

Тип данных `PCDATA` (parseable character data, символьные данные, доступные для анализа) указывает на то, что элемент данных `Internals` может содержать любые данные, с которыми может работать программа-анализатор. Кроме типа `PCDATA` существуют еще два: `EMPTY` и `ANY`. Указание `EMPTY` означает, что элемент данных является пустым, например, `<empty_element/>`, а указание `ANY` — что содержимое элемента данных специально не описывается.

А как описывать элементы данных, включающие другие элементы данных? Рассмотрим описание элемента данных `Room`:

```
<!ELEMENT Room (Externals+, Internals?)>
```

Значения в скобках указывают на элементы данных, допустимые внутри элемента `Room`. Символ `+` после элемента `Externals` означает, что таких элементов внутри элемента `Room` может быть несколько. Если бы этого символа не было, это означало бы, что внутри элемента `Room` *должен* присутствовать элемент `Externals`, причем только один. Символ `?` после элемента `Internals` означает, что внутри элемента `Room` *может* присутствовать один или несколько элементов `Internals`, т. е. наличие элементов `Internals` не является обязательным.

DTD-определение для XML-описания объекта `Room` приводится в листинге 6.3.

Листинг 6.3. DTD-определение для XML-описания объекта `Room`

```
<!ELEMENT Externals (#PCDATA)>
<!ELEMENT Internals (#PCDATA)>
<!ELEMENT Room (Externals+, Internals?)>
```

Сохраним это описание в файле `room.dtd`. Для того, чтобы использовать описание в XML-документе, необходимо добавить в него строку

```
<!DOCTYPE room SYSTEM "room.dtd">
```

Теперь программа-анализатор сможет проверять синтаксис описания объекта `Room`.

Методы описания с помощью формата DTD отличаются чрезвычайной гибкостью и позволяют строго описывать действительно сложные языки разметки, однако, более подробное рассмотрение этих методов выходит за рамки данной книги.

Пространство имен языка XML

Поскольку в языке XML отсутствует описание значений тэгов, одни и те же тэги в разных документах (или даже в разных частях одного документа) могут иметь разный смысл. Для определения смысла элементов данных или атрибутов XML-тэгов в конкретном блоке документа была разработана концепция пространства имен XML (XML namespace). Пространство имен языка XML представляет собой дополнительную спецификацию, определяющую порядок работы с элементами XML-документа. По своим принципам концепция пространства имен напоминает концепцию DTD или более новую концепцию XML-schema. Разница заключается в том, что определение пространства имен позволяет применять определенные правила к отдельным элементам документа.

Пространство имен языка XML задается с помощью атрибута `xmlns`. Значением этого атрибута является ссылка (URI) на соответствующее описание пространства имен. Кроме того, атрибут `xmlns` позволяет указать префикс, который можно использовать перед именем элемента XML-документа для указания пространства имен, которому принадлежит этот элемент.

Рассмотрим пример указания пространства имен для XML-документа.

```
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:mathml="http://www.w3.org/1998/Math/MathML" ...>
```

В этом примере для элемента данных `html` определено два пространства имен, причем одно из них имеет префикс `mathml`. Пространство имен `xhtml` будет применяться по умолчанию к элементу `html` и всем его дочерним элементам, а для того чтобы применить к какому-либо элементу пространство имен `MathML`, следует указать его префикс, например:

```
<mathml:math> ... </mathml:math>
```

Это означает, что элемент `math` и его дочерние элементы принадлежат пространству имен `mathml`.

Язык XHTML

В предыдущем разделе отмечалось, что язык XML можно рассматривать как средство описания языков разметки. Стандарт языка XHTML (Extensible HTML) представляет собой стандарт языка HTML, переформулированный в терминах XML. Сравнивая традиционный язык HTML и язык XML, мы видели, что язык HTML не соответствует многим правилам языка XML, причем, эти несоответствия носят скорее характер "послаблений". Язык XHTML представляет собой вариант языка HTML, следующий строгим правилам синтаксиса языка XML. К преимуществам языка XHTML следует отнести:

- тесную связь с языком XML, и, как следствие, расширяемость;

- возможность использования языка XHTML в устройствах, браузеры которых не обладают достаточными ресурсами для интерпретации неоднозначных конструкций традиционного языка HTML (мобильные телефоны, устройства hand-held);
- возможность совмещения структур *описания* данных и структур *отображения* данных.

Для того чтобы понять различия требований, предъявляемых к синтаксису в языках HTML и XHTML, следует вспомнить требования синтаксиса языка XML, перечисленные в предыдущем разделе. Сравним два документа (листинги 6.4 и 6.5). Первый документ корректен с точки зрения традиционного языка HTML и будет правильно отображен любым браузером, однако он не соответствует требованиям языка XHTML. Второй документ написан в соответствии с требованиями языка XHTML.

Листинг 6.4. Допустимая HTML-страница, не корректная с точки зрения языка XHTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Плохой HTML-документ</TITLE>
</HEAD>
<BODY>
<B><P>Этот документ скорее всего не вызовет проблем при показе в вашем
браузере, однако он не соответствует требованиям XHTML!
<P> Думаю, вам будет нетрудно понять, почему.</B></P>
</body>
</html>
```

Листинг 6.5. Корректный XHTML-документ

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<title>Корректный XHTML-документ</title>
</head>
<body>
<p><b>Этот документ соответствует требованиям XHTML</b></p>
</body>
</html>
```

Обращаем внимание на то, что документ, корректный с точки зрения языка XHTML, не содержит ничего такого, что нарушало бы требования языка HTML. По этой причине любые браузеры, способные корректно отображать документы, написанные в соответствии с требованиями языка HTML 4.1, смогут отображать и XHTML-документы.

Первое, что нужно отметить, — регистр символов в именах тэгов. Имена всех тэгов в языке XHTML должны быть в нижнем регистре. Второе важное отличие языка XHTML заключается в том, что каждый парный тэг XHTML должен быть закрыт. Например, распространенная в языке HTML конструкция (листинг 6.6) недопустима в XHTML.

Листинг 6.6. Фрагмент документа HTML

```
<P>Параграф 1
<P>Параграф 2
<P>Параграф 3
```

Также следует помнить, что закрывающие тэги должны следовать в порядке, противоположном тому, в котором следуют открывающие тэги.

Рассмотрим директиву `<!DOCTYPE>` (листинги 6.4, 6.5), идущую в начале документа. Эта директива указывает на один из трех типов XHTML-документов — Strict, Transitional и Frameset. Тип Strict указывает, что документ строго следует правилам, указанным в описании синтаксиса (`xhtml1-strict.dtd`).

Тип Transitional позволяет внести некоторые послабления в текст XHTML-документа. Этот тип следует использовать для браузеров, не поддерживающих технологию CSS.

Тип Frameset предназначен для организации кадров (frames) при выводе HTML-страниц.

В языке XHTML тэг `<html>` может иметь параметры. Атрибуты `xml:lang` и `lang` позволяют указать язык описаний и документов.

Все значения атрибутов тэгов должны заключаться в кавычки:

```
<font color="#ffffff">
```

Все непарные тэги должны иметь закрывающий символ:

```
<hr/>
```

Среда Borland Kylix позволяет создавать заготовки XHTML-документов для Web-проектов. Для этого нужно выбрать пункт **XHTML Document** на вкладке **Web Documents** диалогового окна **New Items**. Стандартная заготовка XHTML-документа, созданная в Kylix, приводится в листинге 6.7.

Листинг 6.7. Стандартная заготовка XHTML-документа

```
<?xml version="1.0"?>
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title> Untitled1.html </title>
  </head>
  <body>

  </body>
</html>
```

Язык WML

WML (Wireless Markup Language) — язык разметки для работы в Интернете беспроводных устройств (WAP), основанный на языке XML. Для беспроводных устройств (мобильных телефонов, карманных компьютеров и т. п.) язык WML играет ту же роль, что язык HTML (XHTML) для персональных компьютеров.

Почему в случае мобильных устройств возникла потребность в специальном языке разметки? Дело в том, что мобильным устройствам присущ ряд ограничений (малые размеры экрана, узкая полоса пропускания канала связи, ограниченный объем оперативной памяти и т. п.), делающий традиционные языки Web-разметки трудно применимыми. В частности мобильные телефоны не обладают средствами указания (такими, как мышь), поэтому традиционная для языка HTML модель интерфейса point-and-click ("укажи и щелкни") в этих устройствах неприменима.

Сегодня мобильные устройства все чаще подключаются к Интернету, поэтому перед разработчиком сетевых решений возникает задача создания серверного программного обеспечения для работы с языком WML.

Поскольку язык WML основан на языке XML, требования к синтаксису в нем гораздо строже, чем в HTML. Более строгие правила являются преимуществом с точки зрения мобильных устройств, так как позволяют упростить встроенные браузеры.

WML-документ представляет собой набор описаний содержимого экранов-"карт", которые выводятся на экран мобильного устройства по требованию пользователя.

Если WML-документ содержит несколько карт, то первая из них включает в себя список гиперссылок, позволяющих перейти к другим картам документа. Таким образом в языке WML решается проблема прокрутки, когда размеры всего WML-документа превышают возможности экранной памяти. Разумеется, WML-документ может содержать гиперссылки не только на собственные карты, но и на другие WML-документы, а также на некоторые ресурсы других типов, например, графические файлы специального формата wbmp.

Рассмотрим текст простого WML-документа (листинг 6.8).

Листинг 6.8. Документ WML

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
<card id="page1" title="Home Page">
 <p align="center"><strong>This is the first WML page</strong></p>
 <p><do type="accept" label="next page"><go href="#page2"/></do></p>
 <p><do type="prev" label="back"><go href="page3"/></do></p>
</card>

<card id="page2" title="Second Page">
 <p align="center"><strong>This is the second page.</strong></p>
 <a href="other.wml">Other Document</a>
 <do type="accept" label="next page"><go href="#page3"/></do>
 <do type="prev" label="back"><prev/></do>
</card>

<card id="page3" title="Page 2">
 <p align="center"><strong>This is the third page</strong></p>
 <do type="accept" label="next page"><go href="#page1"/></do>
 <do type="prev" label="back"><prev/></do>
</card>
</wml>
```

Тело документа начинается с тэга <wml>, аналогичного тэгу <body> в языке HTML. Поскольку наш документ содержит несколько экранов, между которыми осуществляется переход, каждый тэг <card>, начинающий описание нового экрана, содержит параметр id, идентифицирующий этот экран в рамках данного WML-документа. Параметр id аналогичен по смыслу тэгу

<a name=...> в языке HTML. Кроме параметра id тэг <card> может включать параметр title, описывающий заголовок экрана.

Тэг <p>, отмечающий начало нового параграфа, практически полностью аналогичен соответствующему тэгу в языкеHTML. WML-документ может содержать гиперссылки, описание которых дается так же, как и описание гиперссылок в языке HTML. Нужно учесть только, что язык WML следует общим правилам языка XML и потому больше похож на язык XHTML.

Обычно мобильные устройства позволяют осуществлять управление и навигацию по экранам при помощи нескольких специальных клавиш. Язык WML позволяет назначать определенные действия клавишам устройства. Самый простой способ сделать это — создать новое *действие* (action), которое задается тэгом <do>, и в параметре type указать имя клавиши, функция которой переназначается. В примере из листинга 6.8 мы переназначаем клавиши accept и prev (число возможных клавиш и их мнемоники зависят от модели устройства). Содержимым тэга <do> должна быть инструкция, которую следует выполнить при нажатии на клавишу. В нашем случае это тэг <go>, выполняющий переход к следующей странице, и одиночный тэг <prev/>, выполняющий возврат на предыдущую страницу.

Среда Borland Kylix позволяет создавать заготовки WML-документов. Для этого нужно выбрать пункт **WML Document** на вкладке **Web Documents** диалогового окна **New Items**. Стандартная заготовка WML-документа, созданная в Kylix, приводится в листинге 6.9.

Листинг 6.9. Стандартная заготовка WML-документа

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.2//EN"
 "http://www.wapforum.org/DTD/wml12.dtd">
<wml>
  <card>


  </card>
</wml>
```

Стилевые XSL-шаблоны

В разделе, посвященном описанию языка XML, было сказано, что тэги XML лишь указывают структуру данных, но не то, как эти данные следует отображать. Стилевые XSL-шаблоны позволяют обычным браузерам конвертировать XML-документы в HTML-страницы, оформленные в соответствии с правилами, заданными в шаблоне. Таким образом, XSL-шаблоны реализуют расши-

ряемость языка XML. С помощью языка XML мы можем описывать произвольные структуры данных, а с помощью шаблонов XSL — задавать правила отображения для каждого класса XML-документов. Хотя чаще всего XSL-шаблоны используются для преобразования XML-документов в HTML-страницы, язык XML позволяет использовать при форматировании XML-данных и другие языки разметки, например язык RTF.

Как и другие языки, рассмотренные в этой главе, язык XSL (Extensible Stylesheet Language) основан на языке XML. Следует отметить, что стандарт языка XSL лишь недавно обрел законченную форму, и только последние версии браузеров поддерживают его.

Процесс генерации документа, отображающего XML-данные на основе правил, заданных стилевыми шаблонами, заключается в следующем: при разборе XSL-документа программа-анализатор обрабатывает инструкции этого языка и каждому элементу данных, найденному в XML-структуре, ставит в соответствие набор тэгов, определяющих отображение этого элемента в целевом документе. Процесс генерации HTML-документа из XML-документа при помощи XSL-шаблона показан на рис. 6.2.

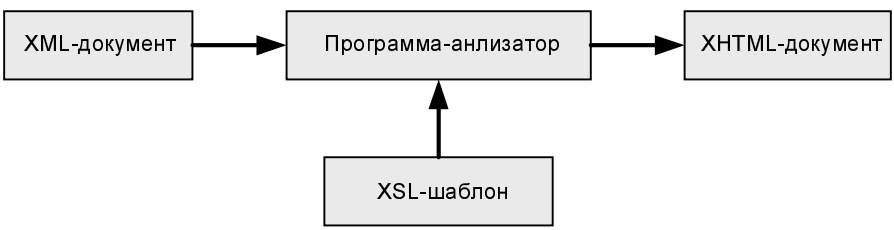


Рис. 6.2. Схема генерации HTML-документа при помощи XSL-шаблона

Язык XSL состоит из трех частей:

- XSLT — язык для преобразования XML-документов;
- XPath — язык определения фрагментов XML-документов;
- XSL Formatting Objects — набор элементов для форматирования XML-документов.

Шаблоны XSL предоставляют следующие возможности:

- преобразование XML-документа в документ XHTML;
- фильтрация и сортировка XML-данных;
- форматирование вывода XML-данных в зависимости от их значения.

Рассмотрим процесс создания и принцип работы XSL-шаблона на конкретном примере.

Напишем XML-документ, содержащий иерархическую структуру данных из листинга 6.1 (листинг 6.10).

Листинг 6.10. XML-документ с описанием структуры объекта `Room`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="room.xsl"?>
<Room>
<Externals>Window</Externals>
<Internals>Table</Internals>
<Externals>Door</Externals>
<Internals>Bed</Internals>
</Room>
```

Как видим, в XML-документ добавлена ссылка на файл room.xsl. Этот файл содержит XSL-шаблон для преобразования XML-документа в формат XHTML. Кроме того, мы специально "перемешали" порядок следования элементов `Internals` и `Externals`. В листинге 6.11 представлен текст XSL-шаблона room.xsl.

Листинг 6.11. XSL-шаблон

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <head>
    <title>Room Description</title>
  </head>
  <body>
    <h2>Room Description</h2>
    <hr/>
    <p><b><font color="#ff0000">Externals</font></b></p>
    <p>
    <ul>
      <xsl:for-each select="Room/Externals">
        <li><xsl:value-of select="."/></li>
      </xsl:for-each>
    </ul>
    </p>
   <hr/>
   <p><b><font color="#0000ff">Internals</font></b></p>
```

```
    <p>
    <ul>
      <xsl:for-each select="Room/Internals">
        <li><xsl:value-of select="."/></li>
      </xsl:for-each>
    </ul>
    </p>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Директива `<xsl:stylesheet ...>` указывает на то, что данный XML-документ является XSL-шаблоном. Далее нам следует указать, к какой части XML-документа применим данный шаблон. Для этого существует директива `<xsl:template ...>` с параметром `match`. Аргументом параметра `match` является выражение языка XPath. Указания фрагментов XML-документа на языке XPath очень похожи на указания элементов файловой системы ОС Linux. Например, выражение `/` указывает на корень документа и означает, что шаблон должен применяться ко всему документу XML.

Далее следуют тэги языка XHTML, которые требуется включить в результирующую страницу. Мы видим, что часть тэгов, определяющих внешний вид XHTML-страницы, никак не связана с данными XML-документа. Тэги XSL-шаблона, предназначенные для обработки XML-данных, имеют префикс `xsl`. Первым таким тэгом, с которым мы сталкиваемся в тексте шаблона, является тэг `<xsl:for-each>`. Этот тэг представляет собой итератор, позволяющий выполнить последовательный перебор всех элементов данных XML-документа, отвечающих определенному критерию. В качестве такого критерия выступает значение параметра `select` тэга `<xsl:for-each>`. Данный параметр позволяет указать имя элементов XML-данных, которые необходимо перебрать. Для указания имени элементов мы опять используем выражения языка XPath. Выражение `Room/Externals` указывает, что нужно перебрать все элементы данных `Externals` элемента `Room`.

Второй тэг XSL-шаблона, с которым мы имеем дело в нашем примере — это тэг `<xsl:value-of .../>` (обратите внимание, что этот тэг — непарный). Данный тэг позволяет добавить в целевую XHTML-страницу значение элемента данных XML. Имя элемента данных, значение которого необходимо добавить в XHTML-страницу, также указывается с помощью параметра `select`, однако имя элемента задается относительно базового имени, указанного в тэге-итераторе. Для итератора `<xsl:for-each select="Room/Externals">`

базовое имя элемента данных — `Room/Externals`. Если бы элемент данных `Externals` содержал другие элементы данных, их имена можно было бы указать в качестве аргументов параметра `select` тэга `<xsl:value-of .../>`. Однако элемент данных `Externals` не содержит дочерних элементов данных, поэтому в целевую XHTML-страницу выводятся данные самого элемента. Как же сослаться на текущий элемент данных в тэге `<xsl:value-of .../>`? Вспомним аналогию между выражениями языка XPath и файловой системой. Элемент данных `Room/Externals` является текущим элементом, значит, в параметре `select` на него можно сослаться как на текущий каталог: `<xsl:value-of select="."/>`.

Далее следуют закрывающие тэги. Поскольку мы имеем дело с документом XHTML, все открытые тэги должны быть закрыты в порядке, противоположном тому, в котором они открывались, а все тэги, не имеющие пары, должны содержать концевой символ `/`, как, например, `<hr/>`.

В листинге 6.12 приводится текст XHTML-страницы, полученной из нашего XML-документа при помощи XSL-шаблона.

Листинг 6.12. Результирующая XHTML-страница

```
<html>
  <head>
     <title>Room Description</title>
  </head>
  <body>
    <h2>Room Description</h2>
    <hr>
    <p><b><font color="#ff0000">Externals</font></b></p>
    <p><ul>
      <li>Window</li>
      <li>Door</li>
    </ul></p>
    <hr>
    <p><b><font color="#0000ff">Internals</font></b></p>
    <p><ul>
      <li>Table</li>
      <li>Bed</li>
    </ul></p>
  </body>
</html>
```

Обратите внимание, что с помощью XSL-шаблона мы не только отформатировали XML-данные, но и произвели их сортировку (сравните порядок перечисления элементов в исходном XML-документе и целевом XHTML-документе).

XSL-шаблоны предоставляют весьма широкие средства в плане форматирования XHTML-документов (и не только), и данный обзор возможностей технологии XSL не претендует на всеохватность. Хорошую документацию и примеры по современным XML-технологиям можно найти на сайте **www.w3schools.org**.

Мы сосредоточим наше внимание на реализации технологии XSL в Borland Kylix. Как и в случае языков, упоминавшихся выше, Kylix позволяет создавать заготовки XSL-документов (для этого нужно воспользоваться той же страницей **Web Documents** диалогового окна **New Items**).

Выше говорилось о том, что для трансформации XML-данных в документ XHTML на основе правил, определенных в XSL-шаблоне, необходима специальная программа-анализатор. В Kylix эту функциональность предоставляет компонент `XSLPageProducer`. Компонент `XSLPageProducer` относится к категории компонентов-генераторов контента, что предполагает использование его в серверных приложениях. В принципе задача преобразования XML-данных в документ XHTML должна возлагаться на браузеры, но выполнение этой операции на стороне сервера может быть оправдано, так как далеко не все из используемых в настоящее время браузеров способны корректно преобразовывать XML-данные на основе шаблонов XSL. В то же время ничто не мешает использовать компонент `XSLPageProducer` в клиентских приложениях, что мы и сделаем в следующем примере.

Сохраним XML-документ из листинга 6.10 под именем room.xml, а XSL-шаблон из листинга 6.11 — под именем room.xsl. Создадим новое приложение Kylix и поместим в главную форму компонент `XSLPageProducer`. Назначим свойству `FileName` этого компонента имя документа-шаблона room.xsl. Компонент `XSLPageProducer` нуждается в источнике XML-данных. В качестве такового может служить компонент `XMLDocument`. Разместим этот компонент в форме приложения и присвоим ссылку на него свойству `XMLData` компонента `XSLPageProducer`. Свойству `FileName` компонента `XMLDocument` присвоим ссылку на файл room.xml. У обоих компонентов есть свойства `Active`, которым нужно присвоить значение `true`.

Для того чтобы отобразить результаты работы компонента `XSLPageProduser`, воспользуемся компонентом `TextViewer`. Назначим обработчик событию `OnFormShow` и добавим в него строку:

```
TextViewer1.Text := XSLPageProducer1.Content;
```

После запуска приложения в окне компонента `TextViewer` должна появиться XHTML-страница, аналогичная той, что показана на рис. 6.3.

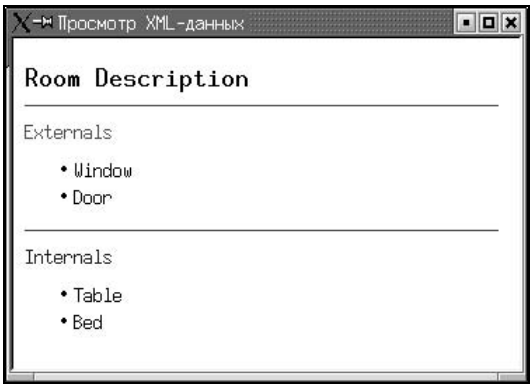


Рис. 6.3. Программа, преобразующая XML-данные в формат XHTML

Объектная модель XML-документов

XML-документы позволяют описывать иерархически организованные структуры данных. С точки зрения программ, обрабатывающих эти документы, очень удобно рассматривать XML-данные как набор иерархических объектов.

У каждого XML-документа есть корневой элемент. Для документа room.xml (листинг 6.10) корневым элементом является элемент `Room`. Обычно компоненты-анализаторы XML-контента позволяют программе получить доступ к корневому элементу и предоставляют программе его имя.

В рамках объектной модели XML-документов каждый элемент данных, начиная с корневого элемента, представляется объектом, обладающим рядом свойств. Некоторые из этих свойств перечислены ниже.

- Свойство `childNodes` — коллекция дочерних элементов данного элемента. Для корневого элемента эта коллекция включает все элементы данных XML-документа. Правила доступа к элементам коллекции зависят от используемого языка программирования, но, как правило, к ним можно обращаться, как к элементам массива.
- Свойство `nodename` — имя элемента данных.
- Свойство `text` — свойство, позволяющее получить доступ к содержимому элемента данных в текстовом формате.

Объектная модель языка XML позволяет не только выполнять анализ иерархии элементов данных документа, но и вносить в них изменения. Для этого в большинстве реализаций объектной модели используется метод `write`.

Уже упоминавшийся компонент `XMLDocument` может использоваться в качестве XML-парсера — специальных программ для извлечения элементов из

HML-документов. Движущей силой компонента `XMLDocument` является синтаксический анализатор объектной модели документа (DOM parser). Kylix предоставляет два возможных движка для синтаксического разбора. Мы в наших примерах будем использовать компонент Xerses XML Parser.

Для представления отдельных элементов XML-иерархии компонент `XMLDocument` использует объект-интрефейс `_di_IXMLNode`. В табл. 6.2 перечислены основные свойства интерфейса `_di_IXMLNode`.

Таблица 6.2. *Свойства интерфейса* `_di_IXMLNode`

Свойство	Тип	Описание
`AttributeNodes`	`_di_IXMLNodeList`	Коллекция атрибутов элемента данных
`Attributes`	`OleVariant`	Перечень атрибутов в неразобранном виде
`ChildNodes`	`_di_IXMLNodeList`	Коллекция дочерних элементов данных данного элемента
`LocalName`	`WideString`	Имя элемента в системе XPath
`NodeName`	`WideString`	Имя элемента данных
`NodeType`	`TNodeType`	Тип элемента данных (элемент, включающий другие элементы, текстовый элемент, атрибут элемента, блок `CDATA` и т. п.)
`NodeValue`	`OleVariant`	Значение элемента данных
`ParentNode`	`_di_IXMLNode`	Родительский элемент данных для данного элемента

Наиболее важные методы интерфейса `_di_IXMLNode` перечислены в табл. 6.3.

Таблица 6.3. *Методы интерфейса* `_di_IXMLNode`

Метод	Возвращаемое значение	Описание
`AddChild`	`_di_IXMLNode`	Добавление дочернего элемента в структуру XML
`DeclareNamespace`	`void`	Установка пространства имен в системе XPath
`FindNamespaceURI`	`_di_IXMLNode`	Поиск элемента данных по его имени в системе XPath

Таблица 6.3 *(окончание)*

Метод	Возвращаемое значение	Описание
HasAttribute	Boolean	Проверка наличия указанного атрибута у элемента данных
HasChildNodes	Boolean	Проверка наличия у элемента данных дочерних элементов
IsTextElement	Boolean	Проверка, является ли данный элемент текстовым
NextSibling	_di_IXMLNode	Следующий элемент данных равного уровня
PreviousSibling	_di_IXMLNode	Предыдущий элемент данных равного уровня
SetAttribute	void	Установка атрибута для данного элемента
SetChildValue	void	Установка значения для дочернего элемента
SetNodeValue	void	Установка значения для текущего элемента
SetText	void	Установка текстового значения для текущего элемента

Свойство DocumentElemnt, являющееся объектом _di_IXMLNode, позволяет получить доступ к корневому элементу. Если XML-документ не имеет корневого тэга, компонентом XMLDocument будет создан элемент Root. Из описания объекта _di_IXMLNode следует, что XML-данные представляются в виде дерева, к которому можно применить алгоритм рекурсивного обхода. В качестве примера использования компонента XMLDocument мы напишем графический анализатор XML-данных, использующий компонент TreeView для вывода информации (для упрощения примера наш анализатор не разбирает секции CDATA).

В заготовке нового проекта разместим компоненты XMLDocument, TreeView, ImageList, OpenDialog и две кнопки — **Открыть** и **Анализировать** (рис. 6.4). Компонент TreeView выложен таким образом, чтобы занимать все свободное клиентское пространство окна приложения, а кнопки размещены на специальной панели (TPanel).

Прежде чем приступать к программированию нашего приложения, следует учесть одну особенность работы компонента XMLDocument в среде Kylix. Ис-

пользуемая версия анализатора объектной модели не очень стабильна. В случае если XML-документ содержит серьезные синтаксические ошибки, приложение может внезапно завершиться в процессе его обработки.

По этой причине при программировании приложений, использующих компонент XML-документ для анализа "ненадежных" документов, следует организовать предварительную проверку синтаксиса XML-документа, благо синтаксические правила XML достаточно просты.

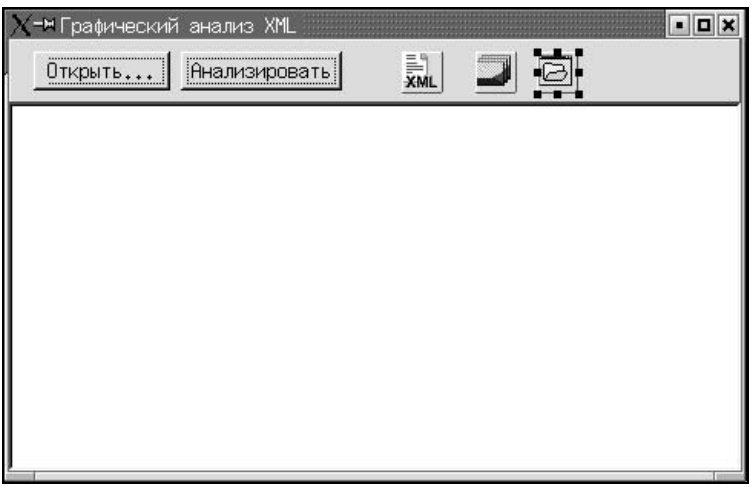


Рис. 6.4. Форма приложения для графического анализа XML-данных

Структура элементов компонента `TreeView` очень похожа на структуру узлов, создаваемую компонентом `XMLDocument` в процессе синтаксического разбора. Самый простой метод заполнения компонента `TreeView` заключается в использовании рекурсивной процедуры.

Заголовочный файл главного модуля приложения-примера приводится в листинге 6.13.

Листинг 6.13. Заголовочный файл приложения графического анализа XML-данных

```
//---------------------------------------------------------------------

#ifndef MainH
#define MainH
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <QControls.hpp>
```

```
#include <QStdCtrls.hpp>
#include <QForms.hpp>
#include <QComCtrls.hpp>
#include <QExtCtrls.hpp>
#include <xercesxmldom.hpp>
#include <XMLDoc.hpp>
#include <xmldom.hpp>
#include <XMLIntf.hpp>
#include <QImgList.hpp>
#include <QDialogs.hpp>

// Константы, указывающие индекс пиктограммы в списке ImageList1
#define PIC_TAG 0
#define PIC_PARM 1
#define PIC_TXT 2

//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TPanel *Panel1;
        TButton *OpenButton;   // Кнопка Открыть...
        TButton *ParseButton;  // Кнопка Анализировать
        TTreeView *TreeView1;
        TImageList *ImageList1;
        TXMLDocument *XMLDocument1;
        TOpenDialog *OpenDialog1;
        void __fastcall ParseButtonClick(TObject *Sender);
        void __fastcall OpenButtonClick(TObject *Sender);
private:      // User declarations
public:       // User declarations
        __fastcall TForm1(TComponent* Owner);
        void FillNodes(_di_IXMLNode xml_node, TTreeNode * tree_node);
};
//---------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------
#endif
```

Предполагается, что объект ImageList1 содержит три пиктограммы для элементов объекта TreeView1 (ссылку на ImageList1 необходимо присвоить свойству Images объекта TreeView1). Пиктограмма с индексом PIC_TAG обозначает тэг, пиктограмма с индексом PIC_PARM — элемент-параметр, а пиктограмма PIC_TXT обозначает текстовый элемент.

В листинге 6.14 приводится основной файл модуля Main.

Листинг 6.14. Основной файл модуля Main

```
//---------------------------------------------------------------------------

#include <clx.h>
#pragma hdrstop

#include "Main.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void TForm1::FillNodes(_di_IXMLNode xml_node, TTreeNode * tree_node)
{
 TTreeNode * cur_node;
  _di_IXMLNodeList attr_list = xml_node->GetAttributeNodes();
  for (int i = 0; i < attr_list->Count; i++)
  {
    String Attr;
    Attr = attr_list->Nodes[i]->NodeName +"="+ attr_list->Nodes[i]->Text;
    cur_node = TreeView1->Items->AddChild(tree_node, Attr);
    cur_node->ImageIndex = PIC_PARM;
    cur_node->SelectedIndex = PIC_PARM;
  }
  if (xml_node->HasChildNodes)
  {
```

```
    tree_node->ImageIndex = PIC_TAG;
    tree_node->SelectedIndex = PIC_TAG;
    _di_IXMLNode cxml_node;
    for (int i = 0; i < xml_node->ChildNodes->Count; i++)
    {
      cxml_node = xml_node->ChildNodes->Nodes[i];
      cur_node = TreeView1->Items->AddChild(tree_node,
      cxml_node->NodeName);
      FillNodes(cxml_node, cur_node);
    }
  }
  else
  {
    tree_node->ImageIndex = PIC_TXT;
    String S = (String) xml_node->Text;
    if (S != "")
    {
      for (int i = 1; i <= S.Length(); i++)
      if (S[i] =='\n')
      {
        S[i] = '\\';
        S.Insert("n", i+1);
      }
      tree_node->Text = S;
    }
    else
    {
      tree_node->Text = xml_node->NodeName;
      tree_node->ImageIndex = PIC_TAG;
    }
    tree_node->SelectedIndex = tree_node->ImageIndex;
  }
}

void __fastcall TForm1::ParseButtonClick(TObject *Sender)
{
  TTreeNode* Node;
  TreeView1->Items->Clear();
```

```
  Node = TreeView1->Items->AddFirst(NULL,
  XMLDocument1->DocumentElement->NodeName);
  FillNodes(XMLDocument1->DocumentElement, Node);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::OpenButtonClick(TObject *Sender)
{
  if (OpenDialog1->Execute())
  {
    XMLDocument1->FileName = OpenDialog1->FileName;
    try
    {
      XMLDocument1->Active = true;
    }
    catch (Exception * e)
    {
      ShowMessage("Анализ файла невозможен.");
    }
  }
}
//---------------------------------------------------------------------------
```

Основную работу в нашем приложении выполняет рекурсивная процедура `FillNodes`. У этой процедуры два параметра. Первый параметр указывает на текущий элемент XML-документа (`xml_node`), второй — на соответствующий ему элемент объекта `TreeView1` (`tree_node`).

Прежде всего, процедура `FillNodes` проверяет, есть ли у текущего XML-элемента атрибуты, и, если есть, добавляет их в качестве дочерних элементов текущего элемента `TreeView1`. Затем проверяется, есть ли у текущего XML-элемента дочерние элементы. Если такие элементы есть, значит, текущий элемент является тэгом. Для каждого дочернего элемента `xml_node` создается новый элемент дерева `TreeView1`, дочерний по отношению к `tree_node`, и затем для каждой пары элементов рекурсивно вызывается процедура `FillNodes`.

Если у текущего XML-элемента нет дочерних XML-элементов, тогда это либо непарный тэг, либо текстовый элемент. Если элемент содержит текст, этот текст будет отображен в объекте `TreeView1`. При этом символы переноса строки, которые в соответствии с правилами языка XML не игнорируются, заменяются наборами символов `\n`.

Процедура `FillNodes` вызывается из обработчика события `OnClick` для кнопки `ParseButton`. Обработчик получает ссылку на корневой элемент XML-документа, создает соответствующий ему корневой элемент компонента `TreeView1` и передает оба параметра процедуре `FillNodes`.

Загрузка нового файла для анализа выполняется обработчиком события `OnClick` для кнопки `OpenButton`. Имя открытого файла присваивается свойству `FileName` объекта `XMLDocument1`. Каждый раз, когда свойству `FileName` присваивается новое значение, свойство `Active` объекта `XMLDocument1` принимает значение `false`. Следует учитывать, что фактический разбор XML-документа производится в ответ на присвоение свойству `Active` значения `true`. Если разбираемый XML-документ содержит ошибки, могут возникнуть исключения, которые и "вылавливает" блок `try... catch`.

Используемый нашей программой XML-анализатор Xerces содержится в библиотеке libxercesxmldom.so.1.0 из каталога kyli3/bin/, которую необходимо распространять вместе с приложением.

Созданное нами приложение можно использовать как для анализа документов, созданных на стандартном языке XML, например, документа room.xml, так и документов, написанных на языках, являющихся XML-производными, например, язык XHTML. Попробуем обработать с помощью нашей программы простой XHTML-документ, текст которого приводится в листинге 6.15.

Листинг 6.15. XHTML-документ test.html

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Test XHTML Page</title>
  </head>
  <body>
    <h2>This is the simple XHTML page</h2>
    <p>Since XHTML is based on <b>XML</b> it is possible to parse it
    ↳with an <u>XML Parser</u>.
    <br/>
    <a href="somedoc.html">This is an anchor</a>
    </p>
  </body>
</html>
```

На рис. 6.5 показано, как этот документ выглядит в окне браузера, а на рис. 6.6 — в окне нашего приложения-анализатора.

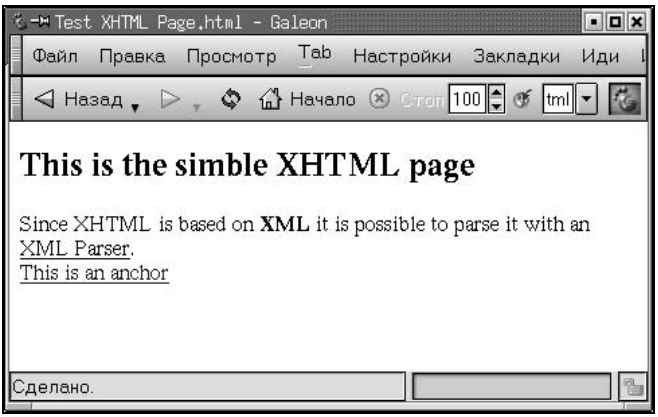


Рис. 6.5. Страница test.html в окне браузера

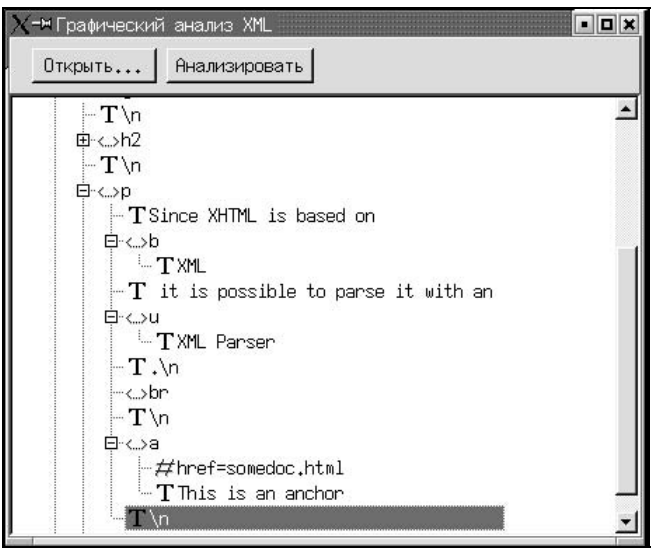


Рис. 6.6. Графический анализ страницы test.html

Использование мастера XML Data Bindings

Компонент `XMLDocument` позволяет анализировать объектную структуру произвольных XML-документов. Интерфейс компонента `XMLDocument` предоставляет лишь базовые средства, которые не очень удобны при работе с конкретным документом. Часто бывают случаи, когда желательно иметь

простой интерфейс программирования для работы с документом XML, имеющим заранее определенную структуру. Особенно полезна такая возможность при программировании в ОС Linux, так как в этой системе огромное количество приложений хранит данные о своих настройках именно в XML-файлах. Создание специализированных интерфейсов программирования для работы с определенными типами файлов XML может потребоваться также для приложений, хранящих большие объемы данных в XML-формате. Формат XML может использоваться в приложениях, подобных приложениям баз данных и в этом случае может быть очень полезно иметь набор классов, позволяющих выполнять основные операции с элементами данных.

В среде Kylix существует возможность автоматически генерировать наборы классов, отражающие объектную модель конкретного XML-документа. Для этой цели служит мастер XML Data Bindings. Внешне работа мастера XML Data Bindings очень напоминает работу мастера **WSDL Importer**, о котором упоминалось в главе 4. При выборе на странице **New** диалогового окна **New Items** пункта **XML Data Bindings**, открывается новое диалоговое окно, в котором нам предлагается ввести имя файла.

В качестве примера создадим интерфейсы для файла bookmarks.xml интернет-браузера Galeon. Если хотя бы раз запустить Galeon, в нашем домашнем каталоге должен появиться подкаталог .galeon, в котором, кроме прочего, мы найдем файл bookmarks.xml, хранящий закладки браузера. Укажем полное имя этого файла в диалоговом окне мастера **XML Data Bindings**, и после нескольких шагов у нас появится модуль `bookmarks`, содержащий объявления и описания следующих классов и интерфейсов (мы пользуемся вариантом Delphi Language IDE):

- `TXMLAutobookmarksType`;
- `TXMLBookmarksType`;
- `TXMLFolderType`;
- `TXMLSiteType`;
- `TXMLSiteTypeList`;
- `TXMLString_List`;
- `IXMLAutobookmarksType`;
- `IXMLBookmarksType`;
- `IXMLFolderType`;
- `IXMLSiteType`;
- `IXMLSiteTypeList`;
- `IXMLString_List`.

Эти объекты полностью отражают модель XML-документа `bookmarks`, а их методы позволяют выполнять различные операции над элементами данных. Кроме перечисленных классов и интерфейсов, в модуле `bookmarks` определены функции, позволяющие создавать новый XML-документ (интерфейс `IXMLBookmarksType`) и загружать XML-данные из файла, хранящегося на диске.

В листинге 6.16 приводится исходный текст процедуры `AddBookmark`, добавляющей закладку для браузера Galeon. В параметре `FileName` процедуре передается имя файла закладок, в параметре `URL` — адрес ресурса, а в параметре `Name` — имя закладки, под которым она войдет в меню **Bookmarks** браузера.

Листинг 6.16. Процедура `AddBookmark`

```
procedure AddBookmark(const FileName, URL, Name : String);
var
  Boookmarks : IXMLBookmarksType;
  NewSite : IXMLSiteType;
  XMLDoc : WideString;
  F : System.Text;
begin
  Boookmarks := Loadbookmarks(FileName);
  NewSite := Boookmarks.Folder.Site.Add;
  NewSite.Url := URL;
  NewSite.Name := Name;
  XMLDoc := Boookmarks.XML;
  System.Assign(F, FileName);
  System.Rewrite(F);
  Write(F, XMLDoc);
  System.Close(F);
end;
```

Глава 7

Быстрая разработка приложений с помощью технологии WebBroker

Эта глава посвящена в основном разработке приложений для Web-сервера с помощью технологии WebBroker, с которой мы уже познакомились (*см. гл. 4*). Как уже отмечалось, главная задача технологии WebBroker состоит в том, чтобы ускорить процесс разработки Web-приложений. Кроме того, технология WebBroker позволяет интегрировать Web-приложения и приложения баз данных, о которых речь пойдет впереди. Заготовка проекта приложения, основанного на WebBroker, создается при выборе пункта **Web-Server Application** на странице **New** диалогового окна **New Items**.

Основа объектной модели приложений WebBroker

Компонент `WebModule` является основой приложений WebBroker и играет в них ту же роль, что и главная форма в обычных приложениях. Компонент `WebModule` выполняет две задачи: во-первых, служит в качестве контейнера для невизуальных компонентов приложения WebBroker (таких, как компоненты-генераторы контента, компоненты для связи с базами данных и др.), а во-вторых, выполняет функции диспетчера входящих HTTP-запросов. В главе 4 говорилось о том, что, как правило, одно CGI-приложение предоставляет несколько сервисов (в терминологии CGI эти сервисы называются actions). При поступлении запроса компонент `WebModule` (при помощи встроенного объекта `WebDispatcher`) определяет, какой из сервисов затребован, и вызывает соответствующий код.

Кроме того, компонент `WebModule` содержит объект `Request` класса `TWebRequest` и `Response` класса `TWebResponse`, позволяющих обрабатывать запросы, поступающие приложению и влиять на формирование ответа приложением. Объект `Request` позволяет получить практически всю информацию о запросе, доступную CGI-приложению *(см. гл. 4)* как из спе-

циальных переменных окружения, так и из стандартного потока ввода. В свою очередь, объект `Response` дает приложению возможность явным образом задавать многие важные параметры ответа. Так, например, свойство `ContentEncoding` позволяет указать кодировку высылаемых HTML-данных, свойство `Cookies` — отправить клиенту "магический" блок информации, а свойства `ReasonString` и `StatusCode` дают приложению возможность должным образом проинформировать клиента об ошибке. Приятной особенностью компонента `Response` является факультативность его использования. При обработке запросов и формировании ответа большинство действий выполняется неявно, с параметрами, принятыми по умолчанию. Обращение к объекту `Response` может потребоваться нам только в том случае, если мы захотим внести изменения в стандартный механизм формирования HTTP-ответа.

Если компонент `WebModule` можно рассматривать как аналог главной формы приложения, то в качестве объекта `Application` в Web-приложениях выступает переменная-объект класса-потомка класса `TWebApplication` (`TApacheApplication` или `TCGIApplication`, в зависимости от типа Web-приложения). Все основные свойства класс `TWebApplication` наследует от класса `TWebRequestHandler`, который призван обрабатывать поступающие запросы. Именно этот класс инкапсулирует механизм взаимодействия с диспетчером запросов и объектами `Request` и `Response`.

Общая схема работы приложения WebBroker выглядит следующим образом: при поступлении запроса на Web-сервер, сервер передает запрос приложению, используя объект `Application`. Объект `Application` формирует объект `WebRequest` и передает его (встроенному или явному) Web-диспетчеру модуля серверного приложения. Диспетчер находит и вызывает соответствующий сервис (action) и возвращает ответ приложения в форме объекта `WebResponse`. Объект `Application` передает ответ Web-серверу.

Выше было сказано, что компонент `WebModule` выполняет функции контейнера невизуальных компонентов и диспетчера вызовов. Если это необходимо, можно "сконструировать" компонент, аналогичный компоненту `WebModule`, из компонентов `DataModule` и `WebDispatcher`. Для того, чтобы создать Web-приложение на основе компонента `DataModule`, нужно всего лишь разместить компонент `WebDispatcher` в форме модуля `DataModule`. Напомним только, что в приложении должен быть только один компонент `WebDispatcher`.

На рис. 7.1 представлена схема работы Web-приложения (обработка запроса `cgiapp/Action1`).

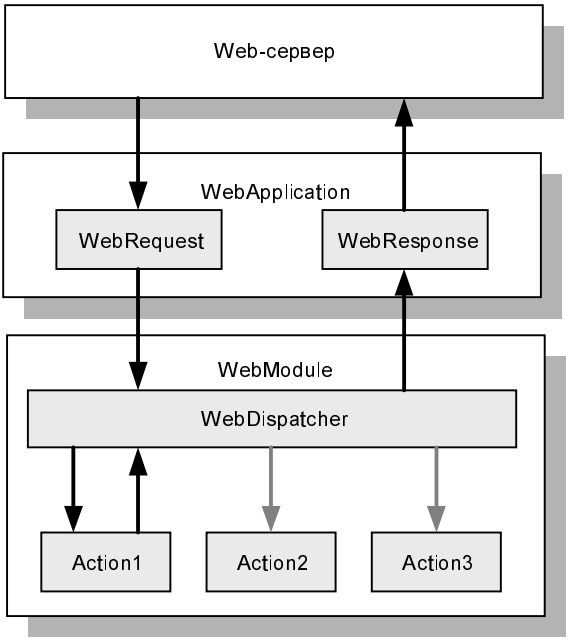


Рис. 7.1. Принцип работы Web-приложения

Для диспетчеризации поступающих запросов используется коллекция `Actions` компонента `WebModule` (на самом деле это свойство принадлежит объекту `WebDispatcher`, который является частью компонента `WebModule`). Элементами коллекции `Actions` являются объекты класса `TWebActionItem`, каждый из которых представляет один сервис данного CGI-приложения. Далее перечислены ключевые свойства класса `TWebActionItem`.

- `PathInfo` — имя, идентифицирующее сервис. Если у приложения `cgiapp` есть сервис с именем `someaction`, обращение к нему обычно выглядит так: /cgi-bin/cgiapp/someaction.
- `MethodType` — это свойство определяет метод, который следует использовать для передачи запроса данному сервису. Возможными значениями этого свойства являются константы: `mtAny` — любой метод; `mtGet` — метод `GET`; `mtHEAD` — метод `HEAD`; `mtPost` — метод `POST`; `mtPut` — метод `PUT`.
- `Default` — это свойство позволяет указать сервис, используемый по умолчанию, т. е., когда при обращении к CGI-приложению имя сервиса не указано.
- `Producer` — ссылка на компонент-генератор контента для данного сервиса. Указывать компонент-генератор в принципе не обязательно. Контент, возвращаемый CGI-приложением, может быть сгенерирован в обработчике события `OnAction`.

Компоненты-генераторы контента

Как же сервис, вызванный объектом WebDispatcher, генерирует ответ приложения? Если свойству Producer объекта TWebActionItem присвоена ссылка на компонент генератор PageProducer, формирование ответа приложением производится автоматически. Считывая страницы шаблонов, компонент-генератор вызывает события OnHTMLTag, как это было описано в главе 4. После того как динамическая страница сформирована, она посылается серверу.

Обработчик события OnHTMLTag позволяет заменять тэги шаблонов значениями, предоставленными программой. Поскольку обработчик является методом объекта WebModule, он может получить доступ к свойствам и методам этого объекта. Возможность обращаться к свойствам Web-модуля существенно расширяет возможности обработчиков событий OnHTMLTag.

Рассмотрим пример, подобный приведенному в главе 4 (см. листинг 4.10), но несколько усложненный. В новом примере обработчик OnHTMLTag обращается к свойству Request, которое содержит данные о поступившем запросе. Приложение GetTime получает запрос, в котором содержится информация о параметрах отображения времени, выбранных пользователем.

В новом CGI-приложении разместим в форме Web-модуля компонент PageProduser и создадим объект класса TWebActionItem со значением свойства PathInfo, равным /, и значением Default, равным true. Свойству Producer присвоим ссылку на объект PageProducer1, а свойству HTMLFile объекта PageProducer1 — ссылку на файл gttmpl.html, текст которого приводится в листинге 7.1.

Листинг 7.1. Шаблон динамической страницы для приложения GetTime

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Текущее Время</title>
  </head>
  <body>
    <p align="Center"><font SIZE="3">
    <table border="2" bgcolor="#e0e0e0">
    <tr>
      <td>
        <p>Текущее время: <#CT></p>
        <p><font color="#000080"><b><#HelloStr></b></font></p>
      </td>
```

```
    </tr>
    </table>
  </font></p>
  </body>
</html>
```

Прежде чем рассматривать обработчик события `OnHTMLTag` для данного шаблона, приведем текст HTML-формы, с помощью которой клиент должен посылать запрос приложению `GetTime` (листинг 7.2).

Листинг 7.2. HTML-форма запроса для приложения GetTime

```
<html>
  <head>
    <title>Выбор формата времени</title>
    <meta content="">
  </head>
  <body>
  <H2>Выбор формата времени</H2>
  <p><form action=http://localhost/cgi-bin/GetTime method=POST>
  <p>Выберите формат представления времени</br>
  <hr/>
  <input type="radio" name=timefmt value="23" checked> 13:00<br/>
  <input type="radio" name=timefmt value="12"> 1:00 PM<br/>
  <input type="checkbox" name="showsecs"> Показывать секунды</br>
  <hr/>
  <input type=submit name=submit>
  </p>
  </form></p>
  </body>
</html>
```

Как мы видим, теперь запрос на отображение динамической страницы, содержащей значение текущего времени, будет содержать дополнительные параметры. Форма содержит элементы **Radio Button** для выбора формата времени и элемент **Check Box**, определяющий, следует ли показывать секунды. Разумеется, параметру `action` следует присвоить значение, соответствующее расположению CGI-приложения в системе пользователя. Внешний вид страницы HTML-формы для отправки запроса показан на рис. 7.2.

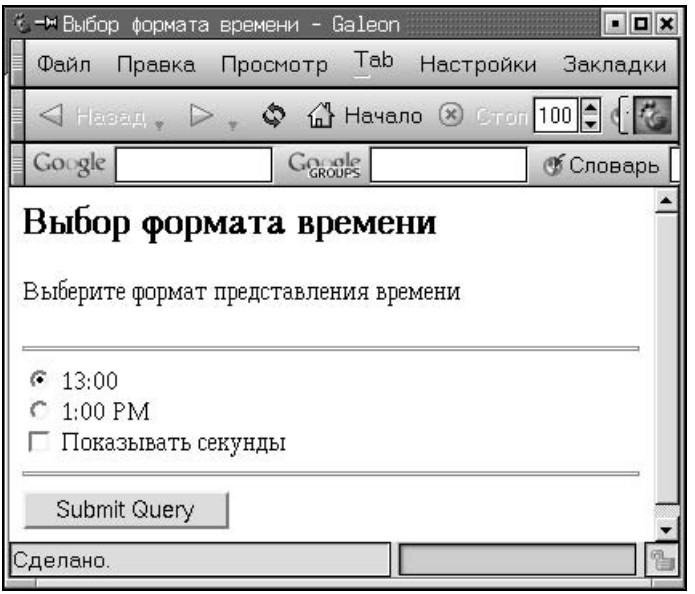


Рис. 7.2. Форма для отправки запроса приложению GetTime

Теперь рассмотрим обработчик события `OnHTMLTag` (листинг 7.3).

Листинг 7.3. Обработчик события `OnHTMLTag`

```
procedure TWebModule2.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
var
  DT : TDateTime;
  Hr, Min, Sec, Msec : Word;
  PreMin, Spec : String;
begin
  DT := Time();
  DecodeTime(DT, Hr, Min, Sec, Msec);
  if TagString = 'CT' then
  begin
    Spec := '';
    if Request.QueryFields.Values['timefmt'] = '12' then
    begin
      if Hr < 13 then Spec := ' AM'
      else
```

```
      begin
        Spec := ' PM';
        if Hr = 0 then Hr := 12
        else Hr := Hr - 12;
      end;
    end;
    if Min < 10 then PreMin := '0'
    else PreMin := '';
    ReplaceText := Format('%d:%s%d', [Hr, PreMin, Min]);
    if Request.QueryFields.Values['showsecs'] <> '' then
    ReplaceText := ReplaceText + ':' + IntToStr(Sec);
    ReplaceText := ReplaceText + Spec;
  end;
  if TagString = 'HelloStr' then
  begin
    if Hr > 12 then
    begin
      if Hr < 16 then ReplaceText := 'Добрый день!'
      else ReplaceText := 'Добрый вечер!';
    end else
    begin
      if Hr > 4 then ReplaceText := 'Доброе утро!'
      else ReplaceText := 'Доброй ночи!';
    end;
  end;
end;
```

При обработке тэга `<#CT>` обработчик анализирует параметры HTTP-запроса при помощи свойства `QueryFields` объекта `WebRequest`. Проверяются значения свойств `timefmt` и `showsecs`. Свойство `QueryFields` является объектом класса `TStrings`. В этом объекте параметры запроса хранятся в "разобранном" виде, в форме списка строк вида `Имя_параметра=Значение`. Такая форма представления параметров позволяет находить их значения при помощи свойства `Values` объекта `QueryFields`. Этому свойству в качестве индексирующего элемента должно быть передано имя параметра. Перечислить имена всех параметров можно при помощи свойства `Names`. Напомню, что способ передачи параметров CGI-приложению зависит от метода, указанного в тэге `<form>`. Свойства `Query` и `QueryFields` объекта `Request` будут содержать параметры CGI-запроса, если в форме запроса использовался метод `GET`. Если же в форме запроса для передачи параметров использовался метод `POST`, неразобранные параметры будут содержать в

свойстве `Content` объекта `Request`, а разобранные — в свойстве `ContentFields`. Если свойству `MethodType` объекта `WebActionItem` было присвоено значение `mtAny`, то клиентская форма может использовать любой метод для отправки параметров CGI-приложению. В этом случае для определения того, какой именно метод использовался, следует проверить значение свойства `MethodType` объекта `Request`.

Пример использования WebBroker: технология Cookies

В силу специфики стандарта CGI и протокола HTTP, приложения, использующие этот стандарт, относятся к категории серверов без сохранения состояния (stateless servers). Это означает, что информация, полученная от пользователя в процессе данной транзакции, не сохраняется в перерывах между транзакциями. Такая архитектура не очень удобна для современных Web-приложений, которые требуют сохранения определенной информации о пользователе на протяжении всего сеанса работы пользователя с приложением (например, данных об авторизации пользователя).

Одним из методов решения этой проблемы является использование Cookies, которые уже неоднократно упоминались.

Суть технологии Cookies заключается в том, что промежуточная информация сохраняется не сервером (обрабатывающим тысячи подобных запросов), а клиентом. В рамках технологии Cookies сервер посылает сохранять на стороне клиента небольшие блоки данных. В зависимости от ситуации программа-клиент может хранить эти блоки данных в оперативной памяти (и тогда они исчезнут вместе с завершением работы программы) либо сохранять на диске в течение некоторого (как правило, определяемого сервером) времени. Если клиент получил от сервера cookie, то при следующем обращении к этому серверу, а возможно и другим серверам из некоторой группы, клиент будет посылать cookie вместе с запросом на передачу страницы. Таким образом, сервер сможет получить некоторую информацию о предыдущих транзакциях, выполненных с этим клиентом.

Замечание

Почему эта технология называется Cookies? Скорее всего такое название навеяно книгой "Алиса в Стране чудес", в которой упоминались "волшебные печенья" (magic cookies).

На практике технология Cookies реализована следующим образом: для того, чтобы отправить клиенту cookie, сервер добавляет в заголовок HTTP-ответа строку:

```
Set-Cookie: NAME=VALUE; expires=DATE; path=PATH;
↳ domain=DOMAIN_NAME; secure; discard; ...
```

За командой `Set-Cookie` следует перечень параметров cookie, разделенных точкой с запятой. `NAME` — произвольное имя cookie, выступающее в роли идентификатора. `VALUE` — строка, содержащая данные, передаваемые клиенту. Эта строка не должна содержать символов точки с запятой, двоеточия и пробела. Пара `NAME=VALUE` является единственным обязательным элементом cookie. Именно эту пару программа-клиент, получившая cookie, будет затем посылать серверу.

В качестве аргумента параметра `expires` выступает значение даты и времени, после наступления, которого cookie следует удалить. Формат даты соответствует стандартам, принятым в Интернете, например: `09-Dec-2002 23:12:40 GMT`.

Параметр `secure` означает, что cookie может передаваться только по соединениям, использующим уровень защиты SSL, а параметр `discard` указывает, что программа-клиент должна удалить cookie при завершении работы.

Параметры `path` и `domain` идентифицируют сервер, посылающий cookie. Параметр `domain` указывает на "доменную" часть в адресе сервера, а параметр `path` — путь к ресурсам на сервере. Путь не обязательно должен быть полным, например запись `path=/;` означает, что данный cookie должен посылаться при запросе любого ресурса с данного сервера. Программа-клиент будет посылать данную запись cookie только при обращении к тем ресурсам, у которых соответствующие части адреса совпадают со значениями параметров `path` и `domain`. Несмотря на важность этих параметров, указывать их явным образом не обязательно. Если параметры `path` и `domain` не будут указаны явно, программа-клиент возьмет их из адреса запроса, вернувшего команду `Set-Cookie`. При этом в параметре `path` будет указан путь к документу (или CGI-сервису), исключая последний элемент. Например, если адрес запроса имеет вид: **site.domain.com/cgi-bin/cgiapp/welcome**, параметру `domain` будет автоматически присвоено значение **site.domain.com**, а параметру `path` — значение **/cgi-bin/cgiapp**.

Один и тот же сервер может послать клиенту несколько cookie, однако тут существуют ограничения. Если количество cookie, посланное сервером клиенту, превышает 20, самый первый из посланных cookie будет удален. Размер одного cookie не должен превышать 4 Кбайта.

Довольно теории. Напишем программу `cookietest`, использующую технологию Cookies для идентификации клиента. У нашей CGI-программы будет три страницы: `register` — форма для идентификации клиента, `welcome` — страница, высылаемая клиенту в случае идентификации, и `vippage` — страница, которую может просматривать зарегистрировавшийся пользователь. Механизм работы программы следующий: незарегистрированному пользователю программа высылает страницу `register`, содержащую форму, в которой клиент должен ввести свое имя. В ответ на отправку формы, если клиент ввел

имя, высылается страница `welcome`, которая устанавливает cookie. После этого клиент может перейти на страницу `vippage`.

Для всех трех страниц нам понадобятся шаблоны, тексты которых приведены в листингах 7.4, 7.5 и 7.6 соответственно.

Листинг 7.4. Шаблон страницы register.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Регистрация</TITLE>
<META http-equiv=Content-Type content="text/html; charset=koi8-r">
</HEAD>
<BODY>
<P>Пожалуйста, зарегистрируйтесь</P>
<FORM action=http://first.intranet/cgi-bin/cookietest/welcome method=get>
<INPUT name=username length="20">
<INPUT type=submit value="Зарегистрироваться" name=submit>
</FORM>
<HR>
</BODY>
</HTML>
```

Листинг 7.5. Шаблон страницы welcome.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Добро пожаловать!</TITLE>
<META http-equiv=Content-Type content="text/html; charset=koi8-r">
</HEAD>
<BODY>
<H2>Добро пожаловать, <#UNAME>!</H2>
<P>Теперь вы можете перейти на эту страницу</P>
<P>
<A href="http://first.intranet/cgi-bin/cookietest/vippage">VIPPage</A>
</P>
</BODY>
</HTML>
```

Листинг 7.6. Шаблон страницы vippage.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Добро пожаловать!</TITLE>
<META http-equiv=Content-Type content="text/html; charset=koi8-r">
</HEAD>
<BODY>
<H2>Здравствуйте, <#UNAME>!;</H2>
<HR>
<P>Вы зарегистрировались и можете просматривать эту страницу!</P>
</BODY>
</HTML>
```

Отметим, что страницы `welcome` и `vippage` используют тэг шаблона `<#UNAME>`. При обработке шаблона этот тэг будет заменен именем, введенным пользователем в форме на странице `register`. Для страницы `welcome` имя пользователя будет взято из параметров CGI-запроса, а для страницы `vippage` — из значения cookie, установленного страницей `welcome`.

В форме Web-модуля приложения мы разместим три компонента `PageProduser`: `RegisterPageProducer`, `WelcomePageProducer` и `WIPPageProducer` (рис. 7.3).

Рис. 7.3. Форма Web-модуля

Компоненты-генераторы должны быть связаны с соответствующими шаблонами страниц. Далее следует создать три объекта `TWebActionItem`, соответствующих сервисам приложения `cookietest`: `register`, `welcome` и `vippage`. Присвоим объектам имена `RegisterAction`, `WelcomeAction` и `VIPAction`. В табл. 7.1 приводятся значения свойств для этих объектов.

Таблица 7.1. *Свойства объектов* `TWebActionItem`

Свойство	Значение
RegisterAction	
Default	True
PathInfo	/
Producer	RegisterPageProducer
WelcomeAction	
Default	False
PathInfo	/welcome
Producer	WelcomePageProducer
VIPAction	
Default	False
PathInfo	/vippage
Producer	WIPPageProducer

Исходные тексты приложения приводятся в листинге 7.7.

Листинг 7.7. Исходные тексты приложения cookietest

```
//---------------------------------------------------------------------
#include "Unit2.h"
#include <WebReq.hpp>
#include <SysUtils.hpp>
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"

TWebModule2 *WebModule2;
//---------------------------------------------------------------------
__fastcall TWebModule2::TWebModule2(TComponent* Owner)
       : TWebModule(Owner)
{
}
```

```
static void initFunc()
{
  WebRequestHandler()->WebModuleClass = __classid(TWebModule2);
}
#pragma startup initFunc 31
//---------------------------------------------------------------------------



void __fastcall TWebModule2::WelcomePageProducerHTMLTag(TObject *Sender,
      TTag Tag, const AnsiString TagString, TStrings *TagParams,
      AnsiString &ReplaceText)
{
  ReplaceText = Request->QueryFields->Values["username"];
}
//---------------------------------------------------------------------------

void __fastcall TWebModule2::VIPPageProducerHTMLTag(TObject *Sender,
      TTag Tag, const AnsiString TagString, TStrings *TagParams,
      AnsiString &ReplaceText)
{
  ReplaceText = Request->CookieFields->Values["UserName"];
}
//---------------------------------------------------------------------------

void __fastcall TWebModule2::WebModuleAfterDispatch(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  String S = Request->PathInfo;
  if (S.Pos("/welcome") != 0)
  {
    TCookie * cookie = Response->Cookies->Add();
    cookie->Name = "UserName";
    cookie->Value = Request->QueryFields->Values["username"];
    Response->SendResponse();
  }
  Handled = true;
}
```

```
//---------------------------------------------------------------------------
void __fastcall TWebModule2::WebModuleBeforeDispatch(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  String S = Request->PathInfo;
  if (S.Pos("/register") != 0)
  {
    if (Request->CookieFields->Values["UserName"] != "")
    {
      Response->SendRedirect((String)Request->ScriptName +
      (String)"/vippage");
        Handled = true;
    }
    Handled = false; // Передача обработки запроса диспетчеру
    return;
  }
  if (S.Pos("/welcome") != 0)
  {
    if (Request->QueryFields->Values["username"] == "")
    {
      if (Request->CookieFields->Values["UserName"] != "")
      {
        Response->SendRedirect((String)Request->ScriptName +
        (String)"/vippage");
      }
      else
      {
        Response->SendRedirect((String)Request->ScriptName +
        (String)"/register");
      }
      Handled = true;
    }
    Handled = false; // Передача обработки запроса диспетчеру
    return;
  }
  if (S.Pos("/vippage") != 0)
  {
    if (Request->CookieFields->Values["UserName"] == "")
```

```
    {
      Response->SendRedirect((String)Request->ScriptName +
      (String)"/register");
      Handled = true;
    }
    Handled = false; // Передача обработки запроса диспетчеру
    return;
  }
}
//---------------------------------------------------------------------------
```

В этой программе мы используем обработчики событий `OnBeforeDispatch` и `OnAfterDispatch`. Эти обработчики вызываются до и после передачи запроса соответствующему объекту `WebActionItem`. Обработчик `OnBeforeDispatch` позволяет изменить порядок диспетчеризации, например, перенаправить вызов. Обработчик `OnAfterDispatch` позволяет добавить дополнительные данные к ответу приложения или изменить этот ответ.

Обработчики событий получают ссылки на объекты `Request` и `Response`. Эти ссылки необходимы, так как объект `WebDispatcher` не всегда используется в составе объекта `WebModule`, а значит объекты `Request` и `Response`, которые принадлежат объекту `WebDispatcher`, могут находиться за пределами области видимости процедуры-обработчика. Кроме того, процедура-обработчик получает параметр-переменную `Handled`, который позволяет указать, следует ли выполнять какие-либо действия по обработке запроса после выхода из процедуры-обработчика, или нет. Если в обработчике `OnBeforeDispatch` присвоить переменной `Handled` значение `false`, дальнейшая обработка запроса будет выполняться стандартными средствами приложения. Если же присвоить этой переменной значение `true`, для программы это будет означать, что обработчик `OnBeforeDispatch` выполнил все необходимые операции, связанные с запросом, и никакие другие действия по обработке запроса приложением выполняться уже не будут, в частности, объект `WebDispatcher` не вызовет соответствующий объект `WebActionItem`. Для обработчика `OnAfterDispatch` переменная `Handled` имеет иной смысл. По умолчанию этой переменной присвоено значение `true`. Если же в обработчике `OnAfterDispatch` присвоить переменной `Handled` значение `false`, ответ просто не будет отправлен клиенту. Для отправки контента из обработчика события `OnBeforeDispatch` или `OnAfterDispatch` следует использовать метод `SendResponse` объекта `Response`. Этот метод формирует HTTP-заголовок ответа и выполняет отправку контента.

В нашем приложении обработчик `OnBeforeDispatch` проверяет, какая страница затребована клиентом. Если это страница `register`, обработчик проверяет, зарегистрирован уже клиент или нет. Если клиент еще не зарегист-

рирован, управление передается диспетчеру запросов, который отправляет клиенту CGI-форму, в противном случае клиент перенаправляется на страницу vippage. Таким образом, от уже зарегистрировавшегося клиента, вызывающего приложение cookietest, не потребуется повторная регистрация.

Показателем того, что пользователь зарегистрирован, служит наличие в запросе клиента cookie UserName. Если cookie входит в состав запроса на получение страницы, значит пользователь уже зарегистрировался. Для проверки отправленных клиентом cookie сервер использует свойство CookieFields объекта Request. Свойство CookieFields является объектом класса TStrings.

Примечание

Напомним, что клиент может передать серверу несколько cookie.

Если клиент затребовал страницу welcome, обработчик OnBeforeDispatch проверяет, содержит ли запрос клиента данные CGI-формы, используя уже знакомое нам свойство QueryFields. Если данные были переданы, управление передается диспетчеру запросов. Если CGI-данные отсутствуют, то возможен один из двух вариантов. Либо пользователь пытается загрузить страницу welcome, не предоставив свое имя для идентификации, либо пользователь уже прошел идентификацию. В первом случае запрос перенаправляется на страницу register, а во втором — на страницу vippage.

При поступлении запроса на загрузку страницы vippage проверяется наличие в запросе cookie UserName. Если cookie отсутствует, выполняется перенаправление клиента на страницу register, в противном случае управление передается диспетчеру, который высылает клиенту страницу vippage.

Cookie UserName посылается клиенту вместе со страницей welcome. Добавление cookie выполняется обработчиком OnAfterDispatch с помощью свойства Cookies объекта Response. Свойство Cookies представляет собой объект класса-потомка класса TCollection и позволяет посылать клиенту несколько cookie одновременно.

Класс TCookie, объекты которого являются элементами коллекции Cookies, содержит свойства, соответствующие параметрам cookie. Обязательными для заполнения являются только свойства Name и Value. Напомним, что при обращении к серверу, клиент, получивший от сервера cookie, будет добавлять в запрос строку вида

```
Cookie: NAME1=DATA1; ...
```

где число элементов соответствует количеству cookie, переданных сервером клиенту. В нашем случае HTTP-запрос клиента будет содержать строку:

```
Cookie: UserName=имя_пользователя;
```

Так что, для получения имени зарегистрированного пользователя мы используем конструкцию Request->CookieFields->Values["UserName"].

В заключение описания работы программы стоит обратить внимание на то, как мы получаем имя программного модуля. Это имя (вместе с частью URL) можно получить из свойства `ScriptName` объекта `Request`. Сгенерированная таким образом ссылка является неполной, так как не содержит имени узла, но Web-браузеры интерпретируют ее правильно.

Обработчик события *OnAction*

До сих пор все наши Web-приложения использовали компоненты-генераторы контента, создающие динамические страницы на основе шаблонов. Подобный способ формирования контента прост, но не универсален. Он не подходит, в частности, для ситуации, когда контент не может быть сформирован на основе шаблона, например, если контент является двоичным файлом.

В какой ситуации приложению может понадобиться генерировать контент не в (X)HTML-форме? Примером может служить программа, обеспечивающая скачивание файлов. На многих сайтах, по разным причинам, скачивание файла допускается только при переходе по ссылке с определенной страницы.

Мы напишем программу, обеспечивающую проверку ссылки при запросе на скачивание файла с помощью обработчика события `OnAction` объекта `WebActionItem`. Это событие вызывается всякий раз, когда диспетчер направляет запрос объекту `WebActionItem`. Обработчик события `OnAction` получает те же параметры, что и обработчики событий `OnBeforeDispatch` и `OnAfterDispatch` объекта `WebDispatcher`: ссылки на объекты `Request` и `Response` и переменную-параметр `Handled`. Переменная-параметр `Handled` в обработчике `OnAction` имеет тот же смысл, что и в обработчике `OnAfterDispatch`. Присвоение этому параметру значение `false` блокирует отправку ответа.

Для нашей программы нам понадобится HTML-страница, содержащая ссылку на загружаемый файл. Текст страницы приводится в листинге 7.8.

Листинг 7.8. Страница download.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Загрузка файла</TITLE>
<META http-equiv=Content-Type content="text/html; charset=koi8-r">
</HEAD>
<BODY>
<H2> Загрузка файла</H2>
```

```
<P>Вы можете загрузить файл <A href="http://first.intranet/↳
cgi-bin/download/archive.zip">archive.zip</A> с этой страницы</P>
</BODY>
</HTML>
```

Наше серверное приложение называется `download`. Создадим объект `WebActionItem` и присвоим его свойству `PathInfo` значение: `/archive.zip`. Текст обработчика `OnAction` для этого объекта приводится в листинге 7.9.

Листинг 7.9. Обработчик события `OnAction`

```
procedure TWebModule2.WebModule2WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  FS : TFileStream;
const
  DownloadPage = '/download.html';
begin
  if Pos(DownloadPage, Request.Referer) <> 0 then
  begin
    Response.ContentType := 'application/octet-stream';
    FS := TFileStream.Create('archive.zip', fmOpenRead);
    Response.ContentLength := FS.Size;
    Response.ContentStream := FS;
    Response.SendResponse;
    FS.Free;
  end
  else Response.SendRedirect(DownloadPage);
  Handled := True;
end;
```

В этом обработчике мы проверяем, с какой страницы поступил запрос на загрузку файла. Для этого используется свойство `Referer`, содержащее значение одноименного поля HTTP-запроса. Содержимое этого поля зависит от клиентской программы. Некоторые браузеры помещают в поле `Referer` полный адрес загружаемого ресурса, тогда как другие — только часть URL, указывающую расположение ресурса на сервере.

Если запрос пришел с нужной страницы, мы устанавливаем тип возвращаемого контента. Для того чтобы программа-клиент не пыталась отобразить полученные данные в своем окне, а сохраняла их на диске, следует присвоить свойству `ContentType` значение: `application/octet-stream`.

Свойства `Response.ContentStream` и `Response.ContentLength` позволяют передавать в качестве контента содержимое потока `TStream`.

Если свойство `Referer` содержит другое значение, выполняется перенаправление клиента на страницу download.html.

Рассмотренный нами пример демонстрирует общую схему отправки не-HTML контента клиенту, однако он обладает одним существенным недостатком. Недостаток заключается в том, что для корректной загрузки файлов каждому файлу следует сопоставить свой объект `WebActionItem` со значением свойства `PathInfo`, совпадающим с именем файла.

В большинстве случаев мы хотим использовать одно то же приложение для отправки клиентам множества различных файлов, имена которых не будут известны заранее.

Для решения этой задачи напишем программу на основе компонента `WebModule`, в которой вообще не будет объектов `WebActionItem`. Все необходимые действия можно выполнить в обработчике события `OnBeforeDispatch`, который получает все запросы, адресованные данному приложению. Назовем наше Web-приложение `getfile`. Строка URL для загрузки файла с помощью приложения `getfile` должна иметь вид: hostname/cgi-bin/getfile/filename, где `hostname` — имя узла сети, а `filename` — имя загружаемого файла. Обработчик события `OnBeforeDispatch` и вспомогательная функция `GetFileRequested` представлены в листинге 7.10.

Листинг 7.10. Обработчик `OnBeforeDispatch` и функция `GetFileRequested`

```
function GetFileRequested(const Req : String) : String;
var
  i : Integer;
begin
  i := Length(Req);
  while (i > 0) and (Req[i] <> '/') do Dec(i);
  SetLength(Result, Length(Req)-i);
  Move(Req[i+1], Result[1], Length(Result));
end;

procedure TWebModule2.WebModuleBeforeDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  FName : String;
  FStream : TFileStream;
```

```
begin
  FName := GetFileRequested(Request.PathInfo);
  if FileExists(FName) then
  begin
    FStream := TFileStream.Create(FName, fmOpenRead or fmShareDenyWrite);
    Response.ContentType := 'application/octet-stream';
    Response.ContentLength := FStream.Size;
    Response.ContentStream := FStream;
  end else
  begin
    Response.StatusCode := 404;
    Response.ReasonString := 'Requested file not found';
    Response.Content :='<HTML><BODY><H2>404 -  file not↳
    found</H2></BODY><HTML>'
  end;
  Response.SendResponse;
  Handled := True;
end;
\
```

Глава 8

Технология WebSnap

Мы уже познакомились с технологией WebSnap (*см. гл. 4*). Данная глава посвящена более подробному описанию возможностей этой довольно сложной технологии. Технология WebSnap расширяет технологию WebBroker в двух направлениях. Во-первых, WebSnap вводит поддержку сценариев на стороне сервера и позволяет получать доступ к объектам серверного приложения из страниц-шаблонов. Во-вторых, WebSnap добавляет готовые решения многих распространенных задач Web-программирования, таких как авторизация пользователей и поддержка сессий. В этой главе мы рассмотрим использование различных компонентов технологии WebSnap на конкретных примерах, но прежде коснемся одного вопроса, не затронутого при предыдущем знакомстве с технологией WebSnap.

Концепция Adapter Actions

Мы уже знаем, что компоненты-адаптеры предоставляют доступ сценариям в шаблонах страниц к данным приложения. Другой важной функцией компонентов-адаптеров является выполнение команд, связанных с HTML-страницами. Команды адаптеров (adapter actions) могут быть связаны с такими элементами HTML-страницы, как компонент ввода типа submit (кнопка) или гиперссылка. Если с нажатием кнопки на HTML-странице связана определенная команда адаптера, то при нажатии на эту кнопку данная команда направляется в составе HTTP-запроса серверному приложению вместе с необходимыми параметрами. Компонент AdapterDispatcher серверного приложения вызовет команду соответствующего адаптера, передав ей необходимые параметры. На уровне программы все команды адаптера представляются объектами класса `TWebActionItem` и его производных.

Вы можете добавлять свои команды в адаптеры точно так же, как мы добавляли поля данных. Код, выполняющий команду, следует вносить в обработчик события `OnAction`.

Как же предоставить пользователю возможность вызвать команду адаптера из полученной им HTML-страницы? Мы помним, что в рамках объектной модели сценариев на стороне сервера компоненты-адаптеры представляются

объектами. Например, компонент-адаптер `EndUserAdapter` доступен в сценарии как объект `EndUser`. Команда этого адаптера `ActionLogout` представляется в сценарии шаблона в виде объекта `EndUser.Logout`. Самый простой способ предоставить пользователю доступ к команде адаптера — воспользоваться свойством `AsHREF` объекта `EndUser.Logout`. Это свойство возвращает ссылку на команду адаптера, которую можно передать серверу, как обычный CGI-запрос. Например, чтобы создать в результирующей странице ссылку **Logout** на команду адаптера `EndUserAdapter.ActionLogout`, в шаблон страницы следует ввести строку:

```
<A HREF=<%=EndUser.Logout.AsHREF%>>Logout</A>
```

Другой вариант обращения к этой команде выглядит так:

```
<A HREF=<%=EndUserAdapter.ActionLogout.AsHREF%>>Logout</A>
```

Иначе говоря, обращаться к командам и методам адаптеров можно, используя их имена в исходном тексте программы.

Итак, запрос на выполнение команды адаптера выглядит как обычный запрос ресурса серверного приложения. В рамках традиционной модели в ответ на такой запрос приложение должно предоставить некий контент, но многие стандартные команды адаптеров не генерируют содержательных HTTP-ответов после выполнения команды. Решить эту проблему можно двумя способами. Первый позволяет связать команду адаптера со страницами в сценарии на стороне сервера. Для этого служит метод `LinkToPage` объекта команды. Этот метод позволяет связать команду адаптера с двумя страницами. Одна страница будет выводиться при успешном выполнении команды, вторая — при неудачном. Например, для команды `EndUser.Logout` мы могли бы написать в сценарии:

```
<% EndUser.Logout.LinkToPage(LogoutSuccess, LogoutFailure) %>
```

Трудно конечно, представить себе, что выполнение команды **Logout** закончится неудачей.

Внутри самой программы организовать вывод контента также несложно. Это можно сделать либо в обработчике события `OnExecute` (для базовой команды), либо в обработчике события `OnAfterExecute` (для встроенной команды адаптера). Для генерации контента можно воспользоваться объектом `Response` соответствующего Web-модуля так же, как мы делали это в приложениях `WebBroker`.

Кроме возможности обращаться к командам адаптеров как к гиперссылкам и элементам HTML-форм, и возможности передачи клиенту контента после выполнения команды, у команд адаптеров есть еще одна особенность, которая делает их весьма полезным и удобным средством техноло-

гии WebSnap. Речь идет о возможности передачи командам адаптеров параметров. Рассмотрим элемент шаблона:

```
<A HREF="<%=SomeAdapter.Command.AsHREF%>&param=value">Command</A>
```

В этой гиперссылке мы передаем команде адаптера параметр `param` со значением `value`. Обратите внимание на использование символа `&` для отделения параметра. Дело в том, что в тексте ссылки на команду адаптера уже присутствует символ `?`, и мы должны указывать параметр как дополнительный.

Обработчики событий команды (объекта `TWebActionItem`) получают список параметров, переданных команде по средствам свойства `Params`, которое имеет тип `TStrings`. В следующем разделе мы рассмотрим довольно изощренный способ использования команд и полей адаптеров.

Программа просмотра изображений

Напишем CGI-приложение, позволяющее просматривать графические файлы, хранящиеся в некотором каталоге на жестком диске. Концепция нашего приложения такова: на диске создается каталог, в котором размещаются графические файлы различных форматов. Наше CGI-приложение позволяет пользователю просматривать изображения из этого каталога. Для загрузки определенного файла пользователь должен направить приложению CGI-запрос, содержащий порядковый номер файла в каталоге (пользователь получает информацию об общем числе файлов в каталоге). В ответ на запрос приложение формирует HTML-страницу, содержащую ссылку на запрошенный графический файл (тэг `<img ...>`), а также некоторые другие элементы (название файла, средства навигации). Поскольку сами графические файлы могут находиться в произвольном каталоге, входящем в файловое пространство Web-сервера, ссылку на файл также представляет собой CGI-запрос, направляемый тому же приложению. Оба CGI-запроса (на формирование HTML-страницы и передачу графического файла) реализуются в виде дополнительных команд адаптера `ApplicationAdapter`, а дополнительная информация (общее число файлов в каталоге, название выбранного файла) передается при помощи полей адаптера. Для понимания принципов работы нашего приложения следует понять одну тонкость. HTML-страница, возвращаемая в ответ на CGI-запрос, содержит ссылку на графический файл, которая сама является CGI-запросом. Чтобы сформировать такую ссылку в самом шаблоне страницы, нам необходимо иметь на уровне шаблона доступ к параметрам запроса, в ответ на который генерируется страница. Для этого служит специальное поле адаптера.

Создадим новый WebSnap-проект. В качестве основы приложения выберем страничный модуль и назовем его `Home` (рис. 8.1).

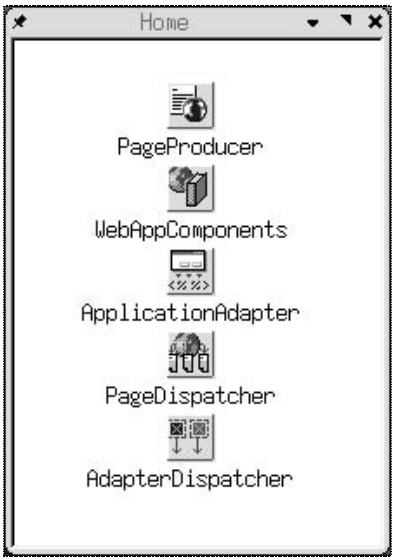


Рис. 8.1. Форма модуля страницы

Наш модуль содержит компонент `PageProducer`, `ApplicationAdapter` и другие компоненты, необходимые WebSnap-приложению. Щелкаем правой кнопкой мыши по компоненту `ApplicationAdapter` и в открывшемся контекстном меню выбираем пункт **Actions Editor...** Открывается окно редактора команд адаптера ApplicationAdapter. В этом окне нужно создать две новые команды (элементы `AdapterAction`). Одну команду мы назовем (присвоив соответствующее значение свойству `Name` в инспекторе объектов) `GetPage`, другую — `ViewImage`. После этого окно редактора должно выглядеть как на рис. 8.2.

Рис. 8.2. Окно редактора команд адаптера ApplicationAdapter

Команда `GetPage` будет служить для вывода HTML-текста страницы приложения, а команда `ViewImage` — для передачи графического файла.

Теперь откроем окно редактора полей адаптера `ApplicationAdapter` (пункт **Fields Editor...** контекстного меню компонента `ApplicationAdapter`) и точно

также создайте три поля типа `AdapterField`. Назовем эти поля `MaxImages`, `ImageFileName` и `ImageNum`. Поле `MaxImages` будет служить для передачи шаблону общего числа файлов в каталоге, поле `ImageFileName` будет хранить имя текущего графического файла, а поле `ImageNum` понадобится для передачи шаблону страницы параметра CGI-запроса.

Обеим командам адаптера передается параметр `img`, значением которого является порядковый номер запрашиваемого графического файла. Шаблон HTML-страницы передает этот параметр запросу на загрузку графического файла.

Примечание

Обычно при использовании технологии WebSnap страница-шаблон создается автоматически. Как же программа "узнает", какой файл шаблона необходимо использовать, ведь соответствующее поле компонента-генератора остается пустым? Имя файла страницы-шаблона совпадает с именем файла, содержащего исходный текст соответствующего модуля (например, файлу Unit1.cpp соответствует шаблон Unit1.html). В языке C++ для связи файла шаблона с модулем используется макрос `USEADDITIONALFILES()`, а в Delphi Language — довольно экзотичная конструкция на основе директивы `$R : {$R *.dfm} {*.html}`. При сохранении файла модуля под другим именем файл шаблона также переименовывается. О том, как размещать файлы шаблонов отдельно от исполнимых файлов WebSnap-приложений (*см. разд. "Компонент LocateFileService" данной главы*).

Исходный текст главного модуля нашей программы приводится в листинге 8.1.

Листинг 8.1. Исходный текст программы просмотра изображений

```
//---------------------------------------------------------------------
#include <clx.h>
#pragma hdrstop

#include "Unit2.h"
#include <WebInit.h>
#include <WebReq.hpp>
#include <WebCntxt.hpp>
#include <WebFact.hpp>

USEADDITIONALFILES("*.html");
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
```

```
#define IMAGE_DIR "/home/andrei/Images/"

THome *Home()
{
  return (THome*)WebContext()->FindModuleClass(__classid (THome));
}

static TWebPageAccess PageAccess;
static TWebAppPageInit WebInit(__classid(THome), caCache, PageAccess <<
wpPublished /* << wpLoginRequired */, ".html", "", "", "", "");

String FindFile(int ind)
{
  TSearchRec FS;
  String FileName;
  String SearchP = (String)IMAGE_DIR + (String)"*";
  FindFirst(SearchP, faAnyFile, FS);
  FindNext(FS);
  for (int i = 1; i <= ind; i++)
  {
    if (FindNext(FS) == 0) FileName = FS.Name;
    else break;
  }
  FindClose(FS);
  return FileName;
}

int THome::GetImageNum(TStrings *Params)
{
  int imnum;
  String S = Params->Values["img"];
  if (S=="") imnum = 1;
  else
  {
    try
    {
      imnum = StrToInt(S);
    }
    catch (Exception * e)
```

```
    {
      imnum = 1;
    }
  }
  return imnum;
}
//---------------------------------------------------------------------

void __fastcall THome::MaxImagesGetValue(TObject *Sender, Variant &Value)
{
  TSearchRec FS;
  String SearchP = IMAGE_DIR;
  int fcount = 0;
  SearchP = SearchP + "*";
  FindFirst(SearchP, faAnyFile, FS);
  FindNext(FS);
  while (FindNext(FS) == 0) fcount++;
  FindClose(FS);
  Value = fcount;
}
//---------------------------------------------------------------------

void __fastcall THome::ImageNumGetValue(TObject *Sender, Variant &Value)
{
  Value = GetImageNum(Request->QueryFields);
}
//---------------------------------------------------------------------
void __fastcall THome::ImageFileNameGetValue(TObject *Sender,
      Variant &Value)
{
  Value = FindFile(GetImageNum(Request->QueryFields));
}
//---------------------------------------------------------------------

void __fastcall THome::ViewImageExecute(TObject *Sender, TStrings
*Params)
{
```

```
  String FileName = FindFile(GetImageNum(Params));
  TFileStream * FStream = new TFileStream((String)IMAGE_DIR +
(String)FileName, fmOpenRead);
  Response->ContentLength = FStream->Size;
  Response->ContentType = "image/"+ExtractFileExt(FileName);
  Response->ContentStream = FStream;
}
//---------------------------------------------------------------------------

void __fastcall THome::GetPageExecute(TObject *Sender, TStrings *Params)
{
  Response->Content = PageProducer->Content();
}
//---------------------------------------------------------------------------
```

Функция `GetImageNum` выполняет служебную роль, она проверяет значение параметра CGI-запроса, переданного ей в качестве аргумента. Целью функции является обработка ситуации, в которой переданный CGI-приложению параметр `img` содержит недопустимое значение.

Функция `FindFile` также является служебной. Эта функция возвращает имя файла, соответствующее переданному ей порядковому номеру. Путь к каталогу, в котором хранятся графические файлы, определяется константой `IMAGE_DIR`.

Процедура `MaxImagesGetValue` — это обработчик события `OnGetValue` для созданного нами поля `MaxImages`. Процедура `ImageNumGetValue` является обработчиком события `OnGetValue` поля `ImageNum`. Эта процедура считывает значение параметра `img`, переданного в составе CGI-запроса команде `GetPage`, и присваивает его полю `ImageNum`. Шаблон страницы воспользуется этим полем для формирования запроса (команды `ViewImage`) на передачу графического файла.

Процедура `ImageFileNameGetValue` заполняет поле `ImageFileName`. Это поле не является необходимым для нашего приложения и носит скорее "декоративный" характер.

Процедура `GetPageExecute` является обработчиком события `OnExecute` команды `GetPage`. Как видим, эта процедура просто посылает клиенту контент, сгенерированный компонентом-генератором `PageProducer` на основе шаблона страницы. Какой же в этом смысл? Дело в том, что в ходе выполнения команды адаптера, поля `ImageNum` и `ImageFileName` принимают значения, соответствующие параметрам команды. При генерации страницы компонентом-генератором эти значения, вызываемые в шаблоне, попадут в результирующий текст HTML-страницы. Иначе говоря, таким способом мы

передаем параметры запроса шаблону, на основе которого создается ответная страница.

Процедура `ViewImageExecute` обрабатывает событие `OnExecute` команды `ViewImage`. Задача этой процедуры — отправка клиенту содержимого графического файла. Также как и предыдущая, эта процедура анализирует значение параметра `img`. Обратим внимание, что для указания типа контента мы используем расширение графического файла.

Смысл обработчиков событий `OnExecute` команд `GetPage` и `ViewImage` должен окончательно проясниться после анализа сценария, входящего в состав страницы-шаблона (листинг 8.2).

Листинг 8.2. Шаблон страницы для программы просмотра изображений

```
<html>
<head>
<META http-equiv=Content-Type content="text/html; charset=koi8-r">
<title>Просмотр изображений</title>
</head>
<%
  imn = ApplicationAdapter.ImageNum.Value;
  if (imn == "") imn = 1
%>
<h2>Файл <%=ApplicationAdapter.ImageFileName.Value%></h2>
<P>
<%
  if (imn != 1)
  {
%>
<A HREF="<%=ApplicationAdapter.GetPage.AsHREF%>&img=<%=imn↳
-1%>">[Предыдущее]</A>
<%
  }
%>
<%
  if (imn != ApplicationAdapter.MaxImages.Value)
  {
%>
<A HREF="<%=ApplicationAdapter.GetPage.AsHREF%>&img=<%=imn↳-
1+2%>">[Следующее]</A>
```

```
<%
  }
%>
</P>
<P>
<IMG SRC="<%=ApplicationAdapter.ViewImage.AsHREF%>&img=<%=imn%>">
</P>
<HR>
<P>
<%
  for (i=1; i<=ApplicationAdapter.MaxImages.Value; i++)
  {
%>
<A HREF="<%=ApplicationAdapter.GetPage.AsHREF%>&img=<%=i%>"><%=i%></A>
<%
  }
%>
</P>
</body>
</html>
```

Обратим внимание на то, как используется значение параметра запроса `img`, переданное в поле `ImageNum` (для удобства это значение присваивается локальной переменной `imn`). Тэг, загружающий изображение, имеет вид:

```
<IMG SRC="<%=ApplicationAdapter.ViewImage.AsHREF%>&img=<%=imn%>">
```

То есть команде адаптера `ApplicationAdapter.ViewImage` передается тот же параметр запроса, что и команде, вызвавшей генерацию страницы.

Значение переменной `imn` можно использовать также для вставки в страницу гиперссылок для перехода к предыдущему и последующему изображению. Необходимость в довольно странной, на первый взгляд, конструкции `imn-1+2` вызвана полиморфизмом переменной `imn`. В тексте сценария переменная `imn` может интерпретироваться и как целочисленная переменная, и как переменная типа "строка". При выполнении операции `imn-1` переменная `imn` интерпретируется как целочисленная, т. е. при `imn=5` результатом `imn-1` будет значение 4, а вот при выполнении сложения эта же переменная интерпретируется как строка, и результатом выражения `imn+1` при `imn=5` будет значение 51.

Используя поле `ApplicationAdapter.MaxImages` и цикл `for`, мы можем создать на результирующей странице последовательность ссылок для произвольного доступа к любому графическому файлу из каталога.

Результат работы программы приводится на рис. 8.3.

Рис. 8.3. Страница, созданная приложением просмотра изображений

Данное приложение наглядно демонстрирует преимущества технологии WebSnap. Само приложение используется только как "средство доставки" контента. Сценарии в шаблоне страницы позволяют нам настраивать внешний вид страницы и методы перехода от одного изображения к другому.

В только что приведенном примере мы использовали команду адаптера для отправки клиенту содержимого графического файла. В принципе, мы могли бы обойтись и без этой команды. Передача динамически сгенерированных графических файлов является столь распространенной задачей при использовании технологии WebSnap, что разработчики ввели специальный тип поля адаптера — `AdapterImageField`. Этот тип поля адаптера отличается от других полей тем, что он скорее похож на команду адаптера, нежели на поле. Поле типа `AdapterImageField` предоставляет ссылку `AsHREF`, к которой можно добавлять параметры, идентифицирующие запрашиваемый графический ресурс.

Можно модифицировать программу просмотра изображений таким образом, чтобы для вывода изображения вместо команды `GetImage` использовалось

поле адаптера. Удалим команду адаптера `GetImage`, а в редакторе полей адаптера `ApplicationAdapter` добавим новое поле типа `AdapterImageField`. В инспекторе объектов изменим имя этого поля на `DynImage`.

Теперь добавим обработчик события `OnGetImage` для данного поля (листинг 8.3).

Листинг 8.3. Обработчик события `OnGetImage`

```
void __fastcall THome::DynImageGetImage(TObject *Sender,
TStrings *Params,
      AnsiString &MimeType, TStream *&Image, bool &Owned)
{
  String FileName = FindFile(GetImageNum(Params));
  TMemoryStream * MS = new TMemoryStream();
  MS->LoadFromFile((String)IMAGE_DIR + (String)FileName);
  MimeType = "image/"+ExtractFileExt(FileName);
  Image = MS;
}
```

Как и обработчик события команды адаптера `OnExecute`, обработчик события `OnGetImage` получает ссылку на список параметров запроса. Параметр `MimeType` позволяет указать тип контента, содержимое которого должно быть передано в объекте, на который указывает параметр `Stream`. Использование потока `TMemoryStream` более предпочтительно по сравнению с `TFileStream`, так как позволяет избежать коллизий, которые могут возникнуть при одновременной обработке запросов, обращающихся к одному и тому же графическому файлу.

Далее нам следует отредактировать сценарий на странице шаблона. Заменим строку

```
<IMG SRC="<%=ApplicationAdapter.ViewImage.AsHREF%>&img=<%=imn%>">
```

на

```
<IMG SRC="<%=ApplicationAdapter.DynImage.Image.AsHREF%>&img=<%=imn%>">
```

Вот и все. С точки зрения функциональности новое приложение просмотра изображений ничем не отличается от предыдущего.

Авторизация пользователей

В предыдущей главе был рассмотрен пример идентификации пользователя с помощью технологии Cookies в рамках технологии WebBroker. В этом разделе мы рассмотрим механизм авторизации пользователей для доступа к отдельным страницам с помощью технологии WebSnap.

Приведенный далее пример демонстрирует возможности визуального программирования. В процессе создания приложения нам придется ввести лишь несколько строчек кода на Delphi Language. На странице **WebSnap** диалогового окна **New Items** выбираем пункт **WebSnap Application**. Открывшееся диалоговое окно позволяет выбрать тип базового модуля нашего приложения. Существует два типа модулей WebSnap: `Page Module` (страничный модуль) и `Data Module` (модуль данных). В качестве основы нашего приложения мы выбираем модуль данных.

Далее мы должны выбрать набор компонентов, которые будет содержать наш модуль. Для выбора компонентов служит диалоговое окно (рис. 8.4), открывающееся при нажатии на кнопку **Components...** . В этом окне мы выбираем компоненты `Application Adapter` (`TApplicationAdapter`), `End User Adapter` (в соответствующем раскрывающемся списке следует выбрать класс `TEndUserSessionAdapter`), `Page Dispatcher` (`TPageDispatcher`), `Adapter Dispatcher` (`TAdapterDispatcher`), `Sessions Service` (`TSessionsService`) и `User List Service` (`TWebUserList`). После нажатия кнопки **OK** появляется окно модуля данных, в котором, кроме выбранных компонентов, содержится еще и компонент `WebAppComponents`.

Компонент `WebAppComponents` регистрирует все компоненты, содержащиеся в модуле (обычно это происходит автоматически). Регистрация компонентов необходима, в частности, для того, чтобы компоненты других модулей (страниц) могли обращаться к компонентам главного модуля.

С компонентом `ApplicationAdapter` мы уже встречались в главе 4. Компонент `EndUserSessionAdapter` предоставляет сценариям в шаблонах объект `EndUser`. Этот объект позволяет управлять регистрацией пользователей и предоставляет информацию об имени зарегистрировавшегося пользователя. Сам компонент `EndUserSessionAdapter` также управляет страницей, содержащей форму для регистрации. Компоненты `PageDispatcher` и `AdapterDispatcher` по своим функциям подобны компоненту `WebDispatcher` технологии WebBroker. Первый из этих компонентов управляет диспетчеризацией страниц, второй — диспетчеризацией вызовов команд адаптеров.

Компонент `SessionsService` позволяет управлять *сессиями* пользователей. Понятие сессии является одним из фундаментальных понятий технологии WebSnap. Каждому HTTP-запросу WebSnap-приложение выделяет собственную сессию. Сессия представляет собой объект, содержащий информацию о сеансе связи, инициированном данным HTTP-запросом. В частности сессия содержит данные о пользователе, пославшем запрос. Сессия может быть завершена пользователем, когда он выбирает опцию **Logout**, либо завершиться автоматически по истечении определенного периода времени. Естественно, все сессии завершаются при завершении работы WebSnap-приложения. К сожалению, при использовании технологии WebSnap в среде ОС Linux возможности использования сессий довольно ограничены (подробнее об этом будет сказано ниже).

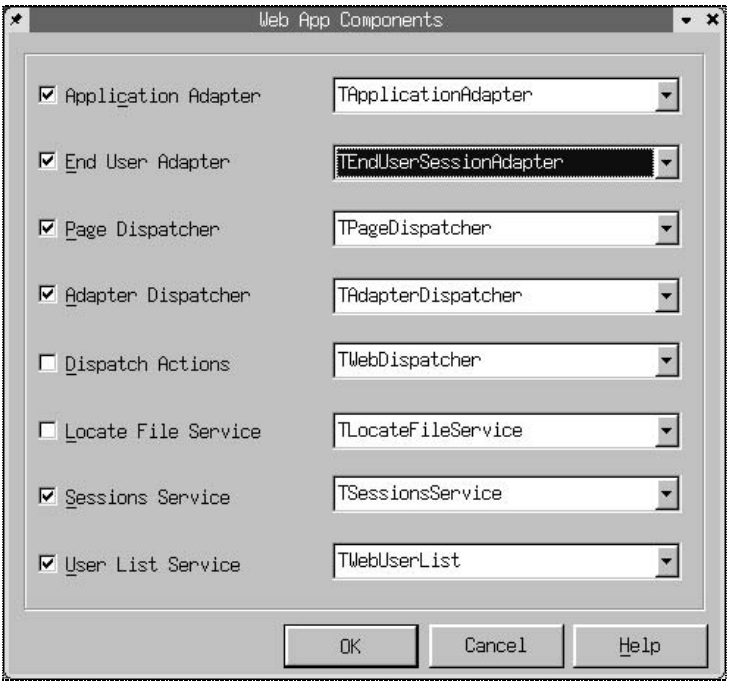


Рис. 8.4. Окно Web App Components, позволяющее выбрать компоненты для проекта, создаваемого пользователем

Компонент `WebUserList` хранит информацию об именах и паролях пользователей, выполняет операции идентификации и авторизации пользователей и предоставляет приложению информацию о событиях, связанных с регистрацией пользователей.

Итак, мы создали модуль, являющийся основой нашего WebSnap-приложения, но у нас нет еще ни одной HTML-страницы, которую бы это приложение могло предоставлять клиенту. Поскольку мы пишем приложение, выполняющее регистрацию пользователей, нашей первой страницей, естественно, должна быть страница, позволяющая пользователю зарегистрироваться. На странице **WebSnap** диалогового окна **New Items** выбираем пункт **WebSnap Page Module**. Открывшееся диалоговое окно позволяет нам установить различные параметры новой страницы, в том числе тип компонента-генератора и имя страницы.

В качестве компонента-генератора в раскрывающемся списке **Type** выбираем компонент `AdapterPageProducer`. Этот компонент необходимо использовать в любом страничном модуле, содержащем компоненты-адаптеры (а создаваемый модуль будет содержать такой компонент).

В строке **Name** в группе **Page** вводим имя страницы: **login**. Данное имя идентифицирует страницу в рамках приложения и никак не связано с именем HTML-страницы-шаблона, создаваемой для генератора контента. По умолчанию страница-шаблон получит то же имя, что и программный модуль, содержащий код для данного страничного модуля.

После завершения работы с диалоговым окном открывается окно нового модуля, содержащее компонент `AdapterPageProducer`. В это окно нам следует добавить компонент `LoginFormAdapter`, который содержит элементы, необходимые для создания HTML-формы регистрации пользователя. Вид окна модуля страницы **login** после добавления компонента `LoginFormAdapter` показан на рис. 8.5.

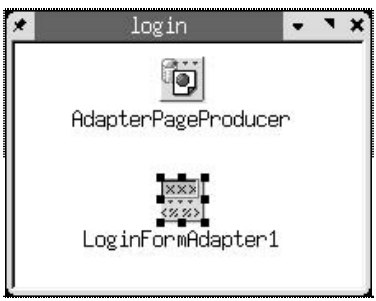


Рис. 8.5. Окно модуля страницы **login**

Теперь нам необходимо настроить компоненты для корректного выполнения операции регистрации и создать HTML-форму в шаблоне страницы модуля **login**. Для этого необходимо предпринять следующие шаги. Дважды щелкаем левой кнопкой мыши по компоненту `AdapterPageProducer` и в открывшемся окне с помощью кнопки **New** создаем объекты `AdapterForm`, `AdapterErrorList`, `AdapterFieldGroup` и `AdapterCommandGroup`. После всех этих действий окно должно выглядеть, как на рис. 8.6. Компонент `AdapterForm1` отвечает за создание HTML-формы регистрации клиента, компонент `AdapterFieldGroup1` создает поля, необходимые для ввода имени пользователя и пароля, а компонент `AdapterCommandGroup1` создает HTML-элемент типа `submit` для отправки информации серверу. Компонент `AdapterErrorList1` служит для вывода на страницу информации об ошибках, возникающих при регистрации. Порядок следования элементов в правой половине окна (рис. 8.6) имеет значение, так как соответствует порядку отображения соответствующих элементов в тексте результирующей HTML-страницы. Теперь с помощью инспектора объектов необходимо связать созданные компоненты между собой. Свойству `Adapter` компонента `AdapterErrorList1` присваиваем ссылку на компонент `LoginFormAdapter1`. То же самое делаем со свойством `Adapter` компонента `AdapterFieldGroup1`. Свойству `DisplayComponent` компонента `AdapterCommandGroup1` присваиваем ссылку на компонент `AdapterFieldGroup1`.

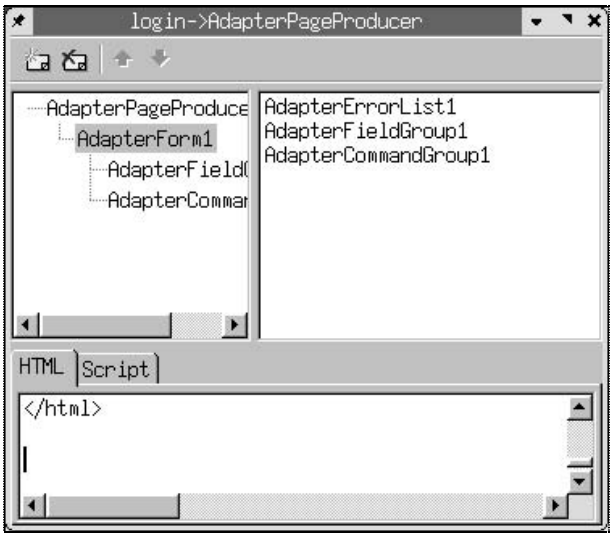


Рис. 8.6. Окно элементов компонента `AdapterPageProducer`

Страница **login** практически готова. Однако мы можем выполнить еще некоторые полезные действия. По умолчанию созданные нами компоненты HTML-формы выводят все надписи на английском языке. Для того, чтобы русифицировать страницу **login**, щелкаем правой кнопкой мыши по компоненту `LoginFormAdapter1` и в открывшемся контекстном меню выбираем пункт **Fields Editor**. В открывшемся окне снова щелкаем правой кнопкой мыши и в контекстном меню выбираем пункт **Add All Fields**. Теперь переходим к инспектору объектов и создаем обработчик для события `OnCreate` объекта Web-модуля страницы **login** (`Tlogin`). Текст обработчика приводится в листинге 8.4 (в нашем случае исходный текст Web-модуля страницы **login** содержится в файле Unit2.pas).

Листинг 8.4. Обработчик события `OnCreate`

```
procedure Tlogin.WebPageModuleCreate(Sender : TObject);
begin
  AdaptUserName.DisplayLabel = "Имя";
  AdaptPassword.DisplayLabel = "Пароль";
  AdaptNextPage.DisplayLabel = "Перейти к";
  ActionLogin.DisplayLabel = "Регистрация";
end;
```

В редакторе исходного текста можно также внести добавления в страницу-шаблон для страницы **login** (в нашем случае эта страница носит имя

Unit2.html). Прежде всего, в раздел HEAD этой страницы следует внести строку:

```
<META http-equiv=Content-Type content="text/html; charset=koi8-r">
```

Можно также в разделе BODY добавить какой-нибудь поясняющий текст.

Работа со страницей **login** закончена, и мы вновь переходим к главному модулю нашего приложения. Прежде всего нам следует сделать страницу **login** страницей, предоставляемой по умолчанию. Для этого мы присваиваем имя страницы (**"login"**) свойству DefaultPage компонента PageDispatcher. Это же значение необходимо присвоить свойству LoginPage компонента EndUserSessionAdapter.

Далее нам нужно создать хотя бы одну учетную запись для регистрации пользователей. Щелкнем мышью в поле свойства UserItems компонента WebUserList. При этом откроется окно редактора учетных записей. Создадим новую учетную запись и присвоим свойству UserName объекта WebUserItem имя пользователя, например, "Andy", а свойству Password — значение "letmein". После этого окно редактирования учетных записей должно выглядеть как на рис. 8.7.

Для того чтобы приложение работало как следует, нужно выполнить еще одно "магическое" действие: щелкнуть правой кнопкой мыши по компоненту EndUserSessionAdapter, в открывшемся контекстном меню выбрать пункт **Fields Editor...**, в открывшемся окне редактора снова щелкнуть правой кнопкой мыши и в контекстном меню выбрать пункт **Add All Fields**.

Рис. 8.7. Редактирование учетных записей

Наше приложение предназначено для того, чтобы предоставлять зарегистрированным пользователям доступ к неким Web-ресурсам. Пришло время создать такой ресурс. Для этого создадим еще одну страницу (пункт **WebSnap Page Module**), в диалоговом окне свойств страницы введем имя

страницы — **VIPPage**, а также отметим флажок **Login Required**, который означает, что для доступа к данной странице необходимо зарегистрироваться. После создания модуля новой страницы можно добавить в текст ее шаблона (Unit3.html) какой-нибудь "очень важный" текст.

Перед завершением работы над приложением нам осталось внести еще одно небольшое изменение. Сценарий, содержащийся в шаблоне страницы **VIPPage**, включает в себя команду **logout**. Обычно эта команда служит для завершения сессии данного пользователя, но в нашем CGI-приложении, которое завершается после передачи клиенту затребованного ресурса и которое будет требовать зарегистрироваться при каждом обращении к странице **VIPPage**, эта команда не имеет смысла. Мы сделаем так, чтобы команда **logout** просто перенаправляла клиента на страницу регистрации. Щелкаем правой кнопкой мыши по компоненту `EndUserSessionAdapter` и в открывшемся контекстном меню выбираем пункт **Actions Editor...** В открывшемся окне редактора щелкаем правой кнопкой мыши и выбираем пункт **Add All Actions**. В правой половине окна редактора выделяем компонент `ActionLogout` и с помощью диспетчера объектов назначаем обработчик для события `OnAfterExecute` этого компонента. Текст обработчика приводится в листинге 8.5.

Листинг 8.5. Обработчик события `OnAfterExecute`

```
procedure TWebAppDataModule1.ActionLogoutAfterExecute(
 Sender : TObject; Params : TStrings);
begin
  Response.SendRedirect(Request.ScriptName + '/login');
end;
```

Теперь наше приложение готово. Сохраним проект под именем LoginDemo, скомпилируем его и скопируем файлы LogIn, Unit2.html и Unit3.html в каталог /cgi-bin/ Web-сервера. Для работы нашего приложения необходима разделяемая библиотека libjs.borland.so, расположенная в каталоге kylix3/lib/. Скопируем этот файл в каталог /usr/lib/.

Теперь в адресной строке браузера можно набрать строку **http://localhost/cgi-bin/LoginDemo/**. Если мы все сделали правильно, в окне браузера должна появиться страница, подобная той, что показана на рис. 8.8.

Для того чтобы получить доступ к странице **VIPPage**, необходимо ввести имя пользователя и пароль, выбрать в списке **Перейти к** страницу **VIPPage** и нажать кнопку **Регистрация**. После этого в окне браузера появится страница, подобная той, что показана на рис. 8.9.

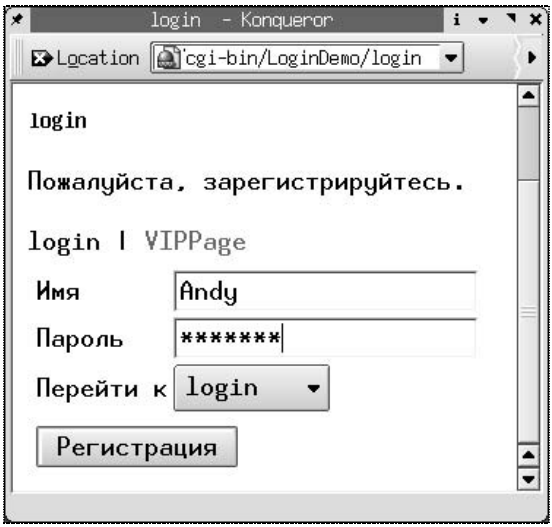


Рис. 8.8. Страница **login** в окне браузера

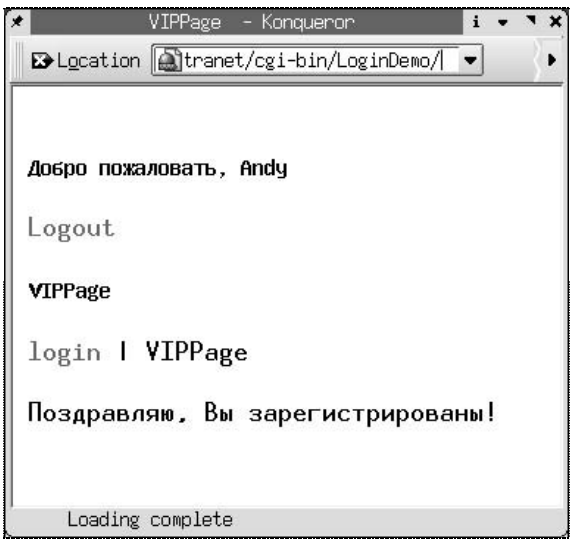


Рис. 8.9. Страница **VIPPage**

Как уже отмечалось, приложение из предыдущего раздела имеет один существенный недостаток: данные в перерывах между транзакциями не сохраняются, и пользователю приходится регистрироваться при каждой попытке загрузить страницу. Проблема связана с тем, что технология WebSnap изначально разрабатывалась в расчете на Windows-интерфейс ISAPI, который позволяет создавать непрерывные сессии. К сожалению, при работе с сер-

вером ОС Linux Apache, такой подход невозможен ни при использовании технологии CGI, ни при использовании разделяемых модулей. В случае использования разделяемых модулей создание непрерывных сессий невозможно потому, что сервер Apache автоматически создает новую сессию при каждом обращении к ресурсу WebSnap-приложения.

К счастью для программистов, работающих с ОС Linux, в случае сервера Apache существуют методы, которые позволяют, если и не создавать полноценные непрерывные сессии технологии WebSnap, то во всяком случае решать многие связанные с этим проблемы.

В среде Delphi Language можно воспользоваться компонентом `CookieSessionsService`, доставшимся среде Kylix 3 "в наследство" от предыдущей версии. Компонент `CookieSessionsService` можно найти в каталоге kylix3/examples/delphi/WebSnap/CookieSessionService/. Скопируем файлы WebCookieSess.pas, WebCookieSessPackage.dpk, WebCookieSessPackage.res и WebCookieSessReg.pas в каталог kylix3/lib/. Откроем в Delphi Language IDE файл WebCookieSessPackage.dpk и в открывшемся окне нажмем кнопку **Compile**. В результате в каталоге kylix3/lib/ появится файл bplWebCookieSessPackage.so. Теперь откроем окно со списком установленных пакетов (команда меню **Component | Install Packages**) и нажмем кнопку **Add**. В открывшемся диалоговом окне выберем файл bplWebCookieSessPackage.so. После этого на странице **WebSnap** палитры компонентов появится новый компонент — `CookieSessionsService`.

Создадим новое WebSnap-приложение аналогично тому, как мы создавали приложение LoginDemo. В форме Web-модуля заменим компонент `SessionServise` компонентом `CookieSessionsService` и присвоим ссылку на новый компонент свойству `Sessions` компонента `WebAppComponents`.

Теперь наше WebSnap-приложение может сохранять информацию о регистрации пользователя в перерывах между загрузкой ресурсов. Правда, команда **Logout** по-прежнему не работает.

Как явствует из названия, компонент `CookieSessionsService` использует схему технологии Cookies для сохранения данных в перерывах между транзакциями. Собственно, это — единственный метод сохранения данных. После выхода из браузера информация о регистрации пользователя не сохраняется, так как используемый компонент `cookie` не сохраняется браузером на диске.

К сожалению, аналога компонента `CookieSessionsService` для языка C++ не существует. Решением проблемы могла бы стать возможность сохранения сессии пользователя по окончании транзакции. Каждая сессия имеет уникальный идентификатор (свойство `SessionID` объекта `Session`). В ответ на поступивший WebSnap-приложению запрос приложение высылает компонент `cookie`, содержащий идентификатор созданной сессии. Этот компонент посылается клиентом приложению при следующем запросе. Если в

версии технологии WebSnap для ОС Linux существовала возможность сохранять, а затем восстанавливать объект `Session`, используя значение `cookie`, можно было бы организовывать непрерывные сессии. Но пока что подобная возможность разработчиками среды Kylix не реализована, и единственная поддержка непрерывных сессий (и то частичная) возможна лишь при использовании Delphi Language IDE и компонента `CookieSessionsService`.

Компонент *LocateFileService*

Все страничные модули приложений WebSnap используют шаблоны страниц. Выше уже говорилось о том, как приложение определяет имя файла шаблона, соответствующего данному модулю. По умолчанию предполагается, что файл шаблона расположен в том же каталоге, что и исполнимый файл приложения. Очевидно, что на практике это неудобно. Исполнимые файлы приложений WebSnap обычно размещаются в каталоге cgi-bin/ (или libexec/, если приложение представляет собой разделяемый модуль сервера Apache). Размещать какие-либо другие файлы в этих каталогах неудобно и не соответствует требованиям безопасности.

Использование компонента `LocateFileService` позволяет размещать шаблоны страниц и другие файлы, необходимые WebSnap-приложениям, в произвольных каталогах.

Компонент `LocateFileService` может быть добавлен в проект в процессе создания WebSnap-приложения (для этого в диалоговом окне **Web App Components**, вызываемом при нажатии на кнопку **Components** в окне **New WebSnap Application**, следует отметить флажок **Locate File Service**). Этот компонент можно добавить также в главный модуль уже созданного WebSnap-проекта (при этом следует убедиться, что ссылка на добавленный компонент назначена свойству `LocateFileService` компонента `WebAppComponents`).

У компонента `LocateFileService` нет каких-либо специфических свойств или методов, но есть три события: `OnFindTemplateFile`, `OnFindIncludeFile` и `OnFindStream`. Если в приложении присутствует компонент `LocateFileService`, одно из его событий вызывается всякий раз, когда какому-либо компоненту приложения требуется найти исходный файл для генерации контента. Событие `OnFindTemplateFile` вызывается, когда какому-либо страничному модулю (или добавленному в приложение компоненту-генератору контента) необходимо найти файл шаблона страницы. Событие `OnFindIncludeFile` вызывается, когда в процессе разбора шаблона страницы генератор встречает указание на включение в шаблон файла сценария. Речь идет о файлах, указанных в основном шаблоне с помощью директивы `<!-- #include file="filename.html" -->`. При обработке шаблона такие файлы включаются в файл шаблона точно также, как include-файлы языка

C++ включаются в файлы исходного текста программы. Важно помнить, что включение файла в шаблон происходит *до* обработки сценария, и таким образом сценарии, содержащиеся во включенных файлах, также будут обработаны. Использование include-файлов в шаблонах дает нам те же преимущества, что и использование include-файлов в языке C++: часто повторяющийся фрагмент кода можно разместить в одном файле, а затем использовать во многих.

Рассмотрим работу обработчиков событий `OnFindTemplateFile` и `OnFindIncludeFile` на простом примере. Создайте специальные каталоги для хранения файлов шаблонов и include-файлов (лучше всего сделать это в каталоге /var/). Назовем эти каталоги соответственно wstemplates и wsincludes. Чтобы у WebSnap-приложений не возникало проблем с правами доступа к файлам, надо разрешить всем пользователям чтение из каталогов:

```
chmod a+r /var/wstemplates
chmod a+r /var/wsincludes
```

Создадим WebSnap-приложение, содержащее компонент `LocateFileService`.

Назначим следующий обработчик событию `OnFindTemplateFile` (листинг 8.6)

Листинг 8.6. Обработчик события `OnFindTemplateFile`

```
void__fastcall TMainModule:: LocateFileServiceFindTemplateFile(
  TObject *ASender, TComponent *AComponent, const AnsiString AFileName,
  AnsiString &AFoundFile, bool &AHandled)
{
  String FileName = (String)"/var/wstemplate/" + (String)AFileName
  if (FileExists(FileName))
  AFoundFile = FileName;
  else
  AHandled = false;
}
```

Параметр `AFileName` содержит имя запрашиваемого файла-шаблона. В строке `AFoundFile` следует вернуть путь к запрошенному файлу, а параметр `AHandled`, по умолчанию равный значению `true`, позволяет указать, была ли найдена запрошенная страница.

Исходный текст обработчика события `OnFindIncludeFile` аналогичен приведенному в листинге 8.6. Следует еще раз отметить, что к компоненту `LocateFileService` обращаются не только страничные модули, но и генераторы контента, добавленные "вручную". Иногда, для того чтобы правильно определить, какой файл шаблона затребован, нам может понадобиться информация о том, какой компонент обращается к компоненту

LocateFileService. Получить эти сведения можно с помощью параметра AComponent обработчика события. Параметру Handled следует присвоить значение false, если обработчик не смог найти требуемый файл.

Скомпилируем приложение и разместим исполнимый файл в соответствующем каталоге Web-сервера, а файлы шаблонов — в созданных подкаталогах каталога /var/. Мы можем убедиться, что модули приложения получают доступ к файлам шаблонов.

Событие OnFindStream предназначено для передачи HTML-контента, который может быть считан из объекта-потока, возможно, созданного объектом-генератором контента. Прототип обработчика такого события приводится в листинге 8.7.

Листинг 8.7 Обработчик события OnFindStream

```
void __fastcall TMainModule::LocateFileServiceFindStream(
  TObject *ASender, TComponent *AComponent, const AnsiString AFileName,
  TStream *&AFoundStream, bool &AOwned, bool &AHandled)
{
  TMemoryStream * MS = new TMemoryStream();
  MS->LoadFromStream(HTMLStream);
  AFoundStream = MS;
}
```

В этом обработчике переменная HTMLStream представляет собой объект-поток, содержащий HTML-данные.

Обработчики OnFindTemplateFile и OnFindStream можно использовать совместно, так чтобы обработчик OnFindTemplateFile обрабатывал одни страницы, а обработчик OnFindStream — другие. Порядок вызова событий следующий: сначала модуль вызывает обработчик OnFindTemplateFile. Если этот обработчик не нашел нужную страницу (в параметре AHandled возвращено значение false), вызывается обработчик OnFindStream, если он, конечно, определен. Если и этот обработчик не может выполнить запрос, модуль ищет шаблон страницы в собственном каталоге.

Допустим, у нас есть два страничных модуля, сохраненных в файлах pmFromFile.cpp и pmFromStream.cpp соответственно. Мы хотим, чтобы модуль, сохраненный в файле pmFromFile.cpp, получал шаблон страницы из файла pmFromFile.html, хранящегося на диске, а модуль, сохраненный в файле pmFromStream.cpp, получал шаблон страницы из потока.

Файл pmFromStream.html следует удалить из проекта, а обработчики событий должны выглядеть следующим образом (листинги 8.8 и 8.9).

Листинг 8.8. Дифференцированный обработчик события OnFindTemplateFile

```
void__fastcall TMainModule:: LocateFileServiceFindTemplateFile(
  TObject *ASender, TComponent *AComponent, const AnsiString AFileName,
  AnsiString &AFoundFile, bool &AHandled)
{
  if (AFileName == "pmFromFile.html")
  {
    AFoundFile = ... // ссылка на файл шаблона pmFromFile.html
    AHandled = true;
    return;
  }
  if (AFileName == "pmFromStream.html")
  {
    AHandled = false;
    return;
  }
}

String FileName = (String)"/var/wstemplate/" + (String)AFileName
  if (FileExists(FileName))
  AFoundFile = FileName;
  else
  AHandled = false;
}
```

Листинг 8.9 Дифференцированный обработчик события `OnFindStream`

```
void __fastcall TMainModule::LocateFileServiceFindStream(
  TObject *ASender, TComponent *AComponent, const AnsiString AFileName,
  TStream *&AFoundStream, bool &AOwned, bool &AHandled)
{
 if (AFileName == "pmFromStream.html")
  {
    AFoundStream = ... // поток, содержащий данные шаблона
    AHandled = true;
    return;
  }
  if (AFileName == "pmFromFile.html")
```

```
  {
    AHandled = false;
    return;
  }
}
```

Технология WebSnap и компонент *WebDispatcher*

Как уже неоднократно говорилось, технология WebSnap является расширением технологии WebBroker. Тем не менее в рассмотренных ранее примерах WebSnap-приложений мы не использовали классических возможностей технологии WebBroker. Для того чтобы задействовать эти возможности, необходимо добавить в WebSnap-проект компонент `WebDispatcher`, расположенный на странице **Internet** палитры компонентов. Сделать это можно как на этапе создания заготовки приложения (для этого в диалоговом окне **Web App Components** нужно отметить флажок **Dispatch Actions**), так и позднее, просто добавив компонент `WebDispatcher` в форму главного модуля приложения.

Получив в свои руки компонент `WebDispatcher`, мы можем задействовать все средства, предоставляемые технологией WebBroker. Создавая новые Web-сервисы (actions) компонента `WebDispatcher`, мы можем использовать компоненты-генераторы контента, добавленные по умолчанию в страничные модули, а также добавлять новые компоненты-генераторы. При использовании уже имеющихся компонентов-генераторов контента можно присваивать значения их свойствам `HTMLFile` и `HTMLDoc`. Это никак не отразится на выводе основных страниц модулей. Разумеется, событие `OnAction` объекта `TWebActionItem` так же, как и события `OnBeforeDispatch` и `OnAfterDispatch` самого компонента `WebDispatcher`, полностью доступны.

Для того чтобы размещать ссылки на сервисы компонента `WebDispatcher`, в Web-сценариях можно использовать поле **Request.ScriptName**. Например, если свойство `PathInfo` объекта `TWebActionItem` имеет значение `"act1"`, то для генерации ссылки на соответствующий сервис в шаблоне-сценарии можно написать:

```
<a href="<%=Request.ScriptName%>/act1">Action1</a>
```

Глава 9

Разработка Web-служб

В двух предыдущих главах мы рассматривали приложения модели клиент-сервер, в которых роль клиента выполнял Web-браузер. В этой и следующей главах речь пойдет о разработке распределенных приложений, различные части которых могут работать под управлением разных операционных систем. Данная глава посвящена разработке приложений Web-служб (Web Services), основанных на протоколе SOAP.

Наше первое знакомство с Web-службами и протоколом SOAP состоялось в главе 4. Напомним, что суть технологии SOAP заключается в том, что клиентские приложения могут получить доступ к объектам приложения-сервера (возможно, выполняющегося на удаленном компьютере) так, как если бы эти объекты были определены в адресном пространстве самого приложения-клиента. Немаловажным соображением при выборе протокола SOAP является и тот факт, что этот протокол лежит в основе технологий .NET, широко внедряемых в настоящее время в мире ОС Windows.

Описание протокола SOAP

В рамках протокола SOAP клиент и сервер обмениваются *сообщениями*. В отличие от других протоколов взаимодействия компонентов распределенных приложений протокол SOAP основан на широко распространенных технологиях HTTP и XML.

Сообщение протокола SOAP представляет собой XML-структуру, содержащую следующие элементы:

- `Envelop` — обязательный элемент, являющийся идентификатором сообщения протокола SOAP;
- `Header` — необязательный элемент, содержащий заголовочную информацию;
- `Body` — обязательный элемент, содержащий данные сообщения;
- `Fault` — необязательный элемент, содержащий информацию об ошибках, возникших в ходе обработки сообщения протокола SOAP.

Сообщения протокола SOAP должны включать указания на пространства имен технологии XML **http://www.w3.org/2001/12/soap-envelope** для элемен-

та `Envelop` и **http://www.w3.org/2001/12/soap-encoding** для указания кодировки сообщения SOAP.

Ниже перечислены синтаксические правила, которым должно следовать сообщение протокола SOAP.

- Синтаксис сообщения должен соответствовать синтаксическим правилам технологии XML.
- Сообщения протокола SOAP должны использовать пространства имен, указанные выше.
- Сообщения протокола SOAP не должны содержать ссылок на формат DTD.
- Сообщения протокола SOAP не должны содержать инструкций по обработке элементов технологии XML.

Общая структура сообщения протокола SOAP представлена в листинге 9.1.

Листинг 9.1. Каркас сообщения протокола SOAP

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Header>
  ...
  ...
</soap:Header>
<soap:Body>
  ...
  ...
  <soap:Fault>
    ...
    ...
  </soap:Fault>
</soap:Body>
</soap:Envelope>
```

Из главы 4 мы уже знаем, что приложение-клиент может вызывать методы объекта, экспортируемого SOAP-сервером. Каким же образом сообщение протокола SOAP используется для передачи запросов и ответов? Рассмотрим конкретный пример. Пусть SOAP-сервер экспортирует интерфейс `IPriceList`, в котором определен метод `GetPrice`:

```
function GetPrice(item : String) : Integer;
```

(мы используем синтаксис Delphi Language). Метод `GetPrice` возвращает значение цены товара, наименование которого передано в параметре `Item`. Допустим, SOAP-клиент, импортирующий интерфейс `IPriceList`, вызывает метод `GetPrice` следующим образом:

```
Price := SOAPImportedPriceList.GetPrice("Eye-nice(tm) Monitor");
```

При этом SOAP-клиент посылает серверу HTTP-запрос, который содержит контент, представленный в листинге 9.2.

Листинг 9.2. Запрос SOAP-клиента

```
<?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><NS1:GetPrice
   xmlns:NS1="urn:Unit2-IPriceList"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <Destination xsi:type="xsd:string">Eye-nice(tm) Monitor</Destination>
</NS1:GetPrice></SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Этот запрос содержит и имя метода, и значение передаваемого параметра. Следует обратить внимание на объявление пространства имен `NS1`. Это пространство имен технологии XML для интерфейса `IPriceList`. Само определение этого пространства имен содержится на сервере (оно было сгенерировано автоматически). Программа клиент ничего не должна знать об этом пространстве имен, кроме ссылки на него (`"urn:Unit2-IPriceList"`). Эта ссылка будет правильно разрешена приложением-сервером.

В ответ на запрос сервер вышлет сообщение протокола SOAP следующего содержания (листинг 9.3).

Листинг 9.3. Ответ SOAP-сервера

```
<?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-ENC:encodingStyle="http://schemas.xmlsoap.org/soap/envelope/">
<NS1:GetPriceResponse
  xmlns:NS1="urn:Unit2-IPriceList">
  <return xsi:type="xsd:int">400</return>
</NS1:GetPriceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Значение, возвращенное методом `GetPrice`, передается в элементе данных `return`. Обратим внимание на то, что в рамках протокола SOAP передается информация о типах параметров и значений. SOAP-приложения могут сравнивать эту информацию о типах с данными, полученными из объявлений соответствующих интерфейсов, и генерировать сообщения об ошибке в случае несовпадения типов. Это свойство протокола SOAP чрезвычайно важно, если учесть, что протокол предназначен для организации взаимодействия между приложениями, написанными на разных языках программирования.

Механизмом, обеспечивающим описание сервисов, предоставляемых приложениями SOAP, не зависящее от языка программирования, является язык WSDL (*см. гл. 4*). Описания сервисов на языке WSDL не только предоставляют информацию об экспортируемых интерфейсах, но и позволяют приложению-клиенту получить сведения о расположении сервера в сети, используемых им портах, имени серверного приложения и имени сервиса, для которого создано WSDL-описание. В примере сообщения-запроса протокола SOAP мы видели, что приложение-клиент использует ссылку на пространство имен технологии XML, определенное на приложении-сервере. Очевидно, что и эту информацию клиентское приложение также получает из WSDL-описаний.

Как и всё в технологии SOAP, язык WSDL основан на языке XML. Тремя основными элементами данных WSDL-документа являются:

- `<portType>` — описывает операции, выполняемые данным сервисом;
- `<message>` — описывает типы сообщений, используемых SOAP-сервисом, число элементов этого типа соответствует числу сообщений, используемых SOAP-сервисом;
- `<types>` — описывает типы данных, используемые SOAP-сервисом;
- `<binding>` — описывает подробности протокола, используемого SOAP-сервисом;
- `<service>` — предоставляет данные о SOAP-сервисе, в том числе URL приложения-сервера.

Схематично структура WSDL-документа представлена в листинге 9.4.

Листинг 9.4. Структура WSDL-документа

```
<?xml version="1.0"?>
<definitions>
  <types>
  ...
  </types>
  <message>
  ...
  </message>
  <message>
  ...
  </message>
  ...
  <portType>
  ...
  </portType>
  <binding>
  ...
  </binding>
  <service>
  ...
  </service>
</definitions>
```

Передача сложных структур данных

В примере (*см. гл. 4*) мы создавали интерфейс, экспортирующий методы, возвращающие значения в простейших форматах. К базовым типам данных SOAP-сервиса можно отнести типы Integer, Boolean, String и различные перечислимые типы. Кроме того, в рамках реализации технологии WebServices от Borland методы интерфейсов SOAP могут возвращать массивы переменной длины, если элементами этих массивов являются значения простых типов.

Методы SOAP-интерфейсов позволяют работать и с более сложными структурами данных, но для того, чтобы задействовать эту возможность при разработке средствами технологии WebServices, придется затратить некоторые дополнительные усилия.

Сложные структуры данных, используемые SOAP-приложениями, должны предоставлять информацию о типах во время выполнения (Runtime Type

Information, RTTI). В рамках технологии WebServices такие структуры следует создавать на основе класса `TRemotable`. Класс `TRemotable` не только способен предоставлять информацию о типах во время выполнения, но также выполнять регистрацию соответствующего нового типа в реестре экспортируемых типов.

Примечание

Термину *remotable*, который широко используется в документации Borland, трудно подобрать русский эквивалент. Будем использовать термин "экспортируемый" при переводе всех терминов Borland, использующих слово "remotable", например: remotable class registry — реестр экспортируемых классов.

Для того чтобы передавать с помощью методов SOAP-интерфейсов сложные структуры данных, необходимо создать класс-потомок `TRemotable`, в котором все поля структуры данных были бы представлены в виде публикуемых (published) свойств.

В качестве иллюстрации механизмов передачи сложных структур данных между компонентами распределенной системы, написанными на разных языках, рассмотрим две программы: SOAP-сервер и SOAP-клиент. Сервер предоставляет интерфейс `IListServ`, позволяющий получить данные из прайс-листа некой компании, торгующей оргтехникой. Клиент отображает эти данные в компоненте `StringGrid`. Программа-сервер будет написана на языке Kylix C++, а программа-клиент — на языке Delphi Language.

Для начала создадим XML-документ catalog.xml, служащий в качестве структурированного хранилища информации о товарах (листинг 9.5).

Листинг 9.5. Файл catalog.xml

```
<?xml version="1.0"?>
<catalog>
<item>
  <name>Eye-nice(tm) Monitor</name>
  <price>400</price>
  <discount>10%</discount>
</item>
<item>
  <name>Red-eyed(tm) Wireless Mouse</name>
  <price>25</price>
  <discount>0%</discount>
</item>
...
</catalog>
```

Корневым элементом в данном документе является элемент `catalog`. Этот элемент содержит дочерние элементы данных `item`, каждый из которых представляет отдельный товар. Элементы `item`, в свою очередь, содержат элементы `name` — наименование товара, `price` — цена и `discount` — скидка.

Раз уж мы выбрали технологию XML в качестве формата хранения данных, неплохо будет создать набор интерфейсов для работы с элементами структуры нашего документа. Воспользуемся для этого уже знакомым нам мастером **XML Data Binding**. После выполнения мастера у нас появятся файлы catalog.h и catalog.cpp, в которых объявляются и описываются следующие интерфейсы (предложенные по умолчанию имена интерфейсов были изменены нами в ходе выполнения мастера **XML Data Binding**, в частности, мы убрали окончание "Type" из имен интерфейсов).

- `_di_IXMLCatalog` — представляет корневой элемент XML-документа. Наиболее важные свойства: `Count` — число дочерних элементов элемента `catalog` и свойство-массив `item`, позволяющее получить доступ к отдельному элементу каталога.
- `_di_IXMLCatalogItem` — представляет отдельный элемент `item`. Для нас наиболее важны методы `Get_name`, `Get_price` и `Get_discount`, возвращающие значения соответствующих элементов.

Теперь можно приступить к написанию приложения-сервера. Выбираем пункт **SOAP Server Application** на вкладке **WebServices** диалогового окна **New Items**. После выбора типа серверного приложения отвечаем отказом на предложение сгенерировать интерфейс для нового модуля. Далее создаем новый модуль данных для серверного приложения (пункт **SOAP Server Data Module**) и в строке ввода **Class Name** вводим `ListServ`. Таким образом, создается новый модуль данных SOAP-сервера, в котором уже есть заготовка интерфейса `IListServ`, экспортируемого сервером. Сохраним проект под именем pricelist и с помощью менеджера проектов добавим в него файл catalog.cpp, созданный мастером **XML Data Binding** на основе файла catalog.xml.

Далее нам следует внести в объявление интерфейса и реализующего его класса методы, с помощью которых клиентское приложение будет получать данные с сервера. В листинге 9.6 приводится (в несколько сокращенном варианте) исходный текст заголовочного файла dmList.h для модуля данных ListServ.

Листинг 9.6. Файл dmList.h

```
//---------------------------------------------------------------------- //
// SOAP Data Module: Interface ListServ
//---------------------------------------------------------------------- //
```

```
#ifndef  dmListH
#define  dmListH

#include <InvokeRegistry.hpp>
#include <Midas.hpp>
#include <SOAPMidas.hpp>
#include <SOAPDm.hpp>
#include <stdio.h>

#ifndef __RPC_FAR
  #define __RPC_FAR
  typedef int LCID;
  #define S_OK 0
  #define E_NOINTERFACE -1
#endif

//  Этот класс является типом, возвращаемым одним из методов интерфейса
// IListServ

class TCatalogItem : public TRemotable {
private:
  AnsiString         FName;
  int                FPrice;
  AnsiString         FDiscount;
__published:
  __property AnsiString Name = { read=FName, write=FName };
  __property int        Price = { read=FPrice, write=FPrice };
  __property AnsiString Discount = { read=FDiscount, write=FDiscount };
};

// Экспортируемый SOAP-интерфейс

__interface IListServ : public IAppServerSOAP
{
public:
  // Эти методы добавлены в заготовку интерфейса программистом
  virtual int  __stdcall getItemsCount()=0;
  virtual TCatalogItem* __stdcall getItem(int Index)=0;
  virtual int  __stdcall getPrice(AnsiString Name)=0;
```

```
};
typedef DelphiInterface<IListServ> _di_IListServ;



// Класс, реализующий интерфейс IListServ

class TListServ : public TSoapDataModule, public IListServ,
public IAppServerSOAP
{
__published:
private:
public:
  __fastcall TListServ(TComponent* Owner);
  virtual int  __stdcall getItemsCount();
  virtual TCatalogItem* __stdcall getItem(int Index);
  virtual int  __stdcall getPrice(AnsiString Name);

  /* IAppServerSOAP */
  ...

  /* IUnknown Methods */
  ...
};


#endif
```

В объявление интерфейса `IListServ` добавлено три метода: `getItemsCount`, `getItem` и `getPrice`. Метод `getItemsCount` возвращает число элементов каталога. Метод `getItem` возвращает данные элемента каталога, указанного индексом, а метод `getPrice` возвращает значение поля `price` для элемента с указанным значением `name`.

Обратим внимание на значение, возвращаемое методом `getItem`. Поскольку каждый элемент каталога содержит несколько элементов данных, результатом метода `getItem` должна быть структура данных с полями, соответствующими этим элементам. В соответствии с требованиями технологии Web-Services мы определяем такую структуру как класс `TCatalogItem`, являющийся потомком класса `TRemotable`, и публикующий все свои свойства в разделе `published`.

Все экспортируемые методы экспортируемого интерфейса (а у него могут быть и не экспортируемые методы) должны быть объявлены как виртуаль-

ные, располагаться в разделе `public` и использовать формат вызова `stdcall`. Точно так же должны объявляться эти методы и в базовом классе `TListServ`.

Реализация методов класса `TListServ` (файл dmList.cpp, в сокращенном варианте) приводится в листинге 9.7.

Листинг 9.7. Файл dmList.cpp

```
//---------------------------------------------------------------------- //
// SOAP Data Module: Interface ListServ
//---------------------------------------------------------------------- //
#include <clx.h>
#pragma hdrstop

#include <InvokeRegistry.hpp>
#include <Midas.hpp>
#include <SOAPMidas.hpp>
#include <SOAPHTTPTrans.hpp>
#include <SOAPDm.hpp>

#include "dmList.h"
#include "catalog.h"

#pragma package(smart_init)
#pragma resource "*.xfm"

#define XML_DOC "/var/XML/catalog.xml"

__fastcall TListServ::TListServ(TComponent* Owner) :
TSoapDataModule(Owner)
{}

// Методы класса TListServ, реализующие методы интерфейса IListServ

int  __stdcall TListServ::getPrice(AnsiString Name)
{
  _di_IXMLCatalog Catalog = Loadcatalog(XML_DOC);
  for (int i = 0; i < Catalog->Count; i++)
  {
    _di_IXMLCatalogItem Item = Catalog->item[i];
```

```
    if ((String)Item->Get_name() == (String)Name)
    {
      int res = Item->Get_price();
      Item->Release();
      Catalog->Release();
      return res;
    }
  }
  Catalog->Release();
  throw new ESOAPHTTPException("No such Item");
}

int  __stdcall TListServ::getItemsCount()
{
  _di_IXMLCatalog Catalog = Loadcatalog(XML_DOC);
  int res = Catalog->Count;
  Catalog->Release();
  return res;
}

TCatalogItem* __stdcall TListServ::getItem(int Index)
{
  _di_IXMLCatalog Catalog = Loadcatalog(XML_DOC);
  if ((Index >= 0) && (Index < Catalog->Count))
  {
    TCatalogItem * CI = new TCatalogItem();
    _di_IXMLCatalogItem Item = Catalog->item[Index];
    CI->Name = (AnsiString)Item->Get_name();
    CI->Price = Item->Get_price();
    CI->Discount = (AnsiString)Item->Get_discount();
    Item->Release();
    Catalog->Release();
    return CI;
  }
  Catalog->Release();
  throw new ESOAPHTTPException("Invalid Index");
}
...
```

```
//************************************************************************ //
// This method will be called by SOAP to create objects to handle
// WebService requests.
//************************************************************************ //
static void __fastcall ListServFactory(System::TObject* &obj)
{
  obj = new TListServ(0);
}

//************************************************************************
// This routines registers the interface and implementation class that
// exposes the interface as a Web Service.
//************************************************************************
void static RegTypes()
{
  InvRegistry()->RegisterInterface(__interfaceTypeinfo(IListServ));
  InvRegistry()->RegisterInvokableClass(__classid(TListServ),
  ListServFactory);
}
#pragma startup RegTypes 32
```

Рассмотрим две автоматически сгенерированные функции — `ListServFactory` и `RegTypes`. Функция `ListServFactory` создает новый экземпляр объекта `TListServ` в ответ на сетевой запрос. Функция `RegTypes` регистрирует экспортируемый интерфейс и класс в реестре экспортируемых объектов. Функция `InvRegistry` возвращает указатель на глобальный объект `TInvokableClassRegistry`. Этот класс позволяет регистрировать экспортируемые классы и интерфейсы.

Для получения данных из XML-документа мы пользуемся интерфейсами, сгенерированными мастером **XML Data Binding** (следует обратить внимание на то, что файл catalog.h включен в файл dmList.cpp). О том, как работать с интерфейсами, представляющими объектную модель XML-данных, говорилось в главе 6.

Метод `getItem` динамически создает новый объект класса `TCatalogItem`. Нам не нужно беспокоиться об уничтожении этого объекта. Сервер SOAP уничтожит его сам после того, как ответ будет отправлен клиенту. Если же сервер SOAP является CGI-приложением, мы вообще можем не уничтожать созданные экземпляры объектов, так как после выполнения запроса процесс завершится и все его ресурсы будут уничтожены автоматически.

Теперь мы напишем приложение-клиент. Для этого приложения нам понадобится описание экспортируемого интерфейса `IListServ` на языке WSDL.

О том, как получить такое описание, говорилось *в* главе 4. Не забудем, что Web-сервер должен прослушивать порт 8080. Получив WSDL-описание, мы с помощью мастера **WSDL Importer**, сможем создать объявление интерфейса `IListServ` на языке Delphi Language.

Приложение-клиент SOAP является обычным приложением среды Kylix. Создадим такое приложение в Kylix Delphi IDE, разместим в его главной форме компоненты `HTTPRIO`, `Button` (присвоим свойству `Name` этого компонента значение "RefreshButton", а свойству `Caption` — значение "Обновить") и компонент `StringGrid` для отображения данных. Свойству `WSDLLocation` объекта `HTTPRIO1` присвоим ссылку на WSDL-файл, описывающий интерфейс `IListServ`, и включим модуль с объявлением этого интерфейса в раздел `uses` главного модуля нашего проекта. Мы можем также, хотя это и не обязательно, установить значения свойств `Port` и `Service` компонента `HTTPRIO1`.

Исходный текст главного модуля приводится в листинге 9.8.

Листинг 9.8. Главный модуль приложения-клиента SOAP

```
unit Main;

interface

uses
  SysUtils, Types, Classes, Variants, QTypes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls, IListServ1, Rio, SOAPHTTPClient, QGrids,
  QExtCtrls;

type
  TForm1 = class(TForm)
    HTTPRIO1: THTTPRIO;
    StringGrid1: TStringGrid;
    Panel1: TPanel;
    RefreshButton: TButton;
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure RefreshButtonClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
    ListServ : IListServ;
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.FormShow(Sender: TObject);
begin
ListServ := HTTPRIO1 as IListServ;
  StringGrid1.Cells[0, 0] := 'Номер';
  StringGrid1.Cells[1, 0] := 'Наименование';
  StringGrid1.ColWidths[1] := 200;
  StringGrid1.Cells[2, 0] := 'Цена';
  StringGrid1.Cells[3, 0] := 'Скидка';
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ListServ := nil;
end;

procedure TForm1.RefreshButtonClick(Sender: TObject);
var
  CA : TCatalogItem;
  i : Integer;
begin
  StringGrid1.RowCount := ListServ.getItemsCount+1;
  for i := 1 to StringGrid1.RowCount-1 do
  begin
    StringGrid1.Cells[0, i] := IntToStr(i);
    CA := ListServ.getItem(i-1);
    StringGrid1.Cells[1, i] := CA.Name;
    StringGrid1.Cells[2, i] := IntToStr(CA.Price);
    StringGrid1.Cells[3, i] := CA.Discount;
```

```
    CA.Free;
  end;
end;
end.
```

В обработчике `OnFormShow` мы инициализируем переменную `ListServ`, являющуюся экземпляром интерфейса `IListServ`, объявленного в модуле `IListServ1`. Основные операции выполняются в обработчике события `OnClick` объекта `RefreshButton` (`RefreshButtonClick`). Обратим внимание, что мы должны явным образом уничтожать объект, возвращенный методом `ListServ.getItem`. На рис. 9.1 показано работающее приложение-клиент.

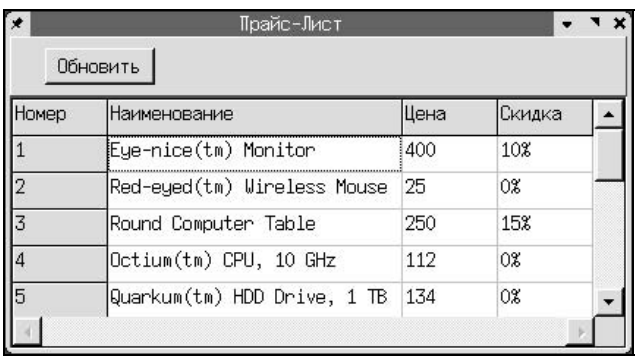


Рис. 9.1. Приложение-клиент SOAP

Каждый раз, когда приложение-клиент SOAP вызывает метод интерфейса, экспортируемого сервером, генерируются HTTP-запрос и HTTP-ответ, содержащие сообщения протокола SOAP. Очевидно, что сетевой трафик (а это, как правило, самое "узкое" место в распределенных системах) будет тем меньше, чем меньше приложение обращается к удаленным интерфейсам.

Наше приложение, безусловно, не является оптимальным с этой точки зрения. Методы удаленного интерфейса вызываются для получения информации о каждом элементе каталога, а ведь таких элементов могут быть сотни! Существует несколько методов решения этой проблемы, и самый простой из них заключается в использовании динамических массивов. Это может показаться странным, но передавать данные в динамических массивах по протоколу SOAP легче, чем в статических. Сократить нагрузку на сеть с помощью динамических массивов можно следующим образом: создавать динамические массивы для каждого типа полей элементов каталога, т. е. динамические массивы значений `name`, `price` и `discount`. Число элементов в каждом массиве будет соответствовать числу элементов в каталоге.

В качестве примера использования динамического массива в нашей распределенной системе рассмотрим передачу с помощью массива значений `name` элементов каталога. Внесем в файл dmList.h определение типа динамического массива, элементами которого являются переменные типа `AnsiString`:

```
typedef DynamicArray<AnsiString> StringArray;
```

и добавим в объявление интерфейса и сопутствующего класса новый метод `getNames`:

```
virtual void __stdcall getNames(StringArray &Names)
```

Реализация метода `getNames` приводится в листинге 9.9.

Листинг 9.9. Метод getNames

```
void __stdcall TListServ::getNames(StringArray &Names)
{
  _di_IXMLCatalog Catalog = Loadcatalog(XML_DOC);
  Names.set_length(Catalog->Count);
  for (int i=0; i<Names.Length; i++)
  {
    _di_IXMLCatalogItem Item = Catalog->item[i];
    Names[i] = Item->Get_name();
    Item->Release();
  }
  Catalog->Release();
}
```

Аналогичные определения и методы можно реализовать и для элементов `price` и `discount`. Конечно, после этого придется импортировать новое WSDL-описание и сгенерировать новые интерфейсы в среде Delphi Language. Реализация импортированного метода `getNames` на языке Delphi Language выглядит следующим образом:

```
procedure getNames(var Names: TStringDynArray); stdcall;
```

Как видим, в модуле, импортирующем интерфейс `IListServ`, тип динамического массива `StringArray` замещен типом `TStringDynArray`, определенным в модуле Types.pas. Заголовочный файл Types.hpp не содержит такого типа, поэтому нам и пришлось явным образом объявлять тип динамического массива в модуле, написанном на языке C++.

Листинг 9.10 демонстрирует использование метода `getNames` в программе-клиенте.

Листинг 9.10. Использование метода `GetNames`

```
var
  LS : IListServ;
  Names : TStringDynArray;
  i : Integer;
begin
  LS := HTTPRIO1 as IListServ;
  LS.getNames(Names);
  for i := 0 to Length(Names)-1 do
  NamesMemo.Lines.Add(Names[i]);
  LS := nil;
end;
```

Кроме типа `TStringDynArray` в модуле Types.pas объявляются аналогичные типы динамических массивов для всех типов данных, приводимых к целочисленным, и всех поддерживаемых в языке Delphi Language типов данных с плавающей точкой.

Так, например тип языка C++

```
typedef DynamicArray<int> IntArray;
```

будет замещен при импорте интерфейса в Delphi Language типом `TIntegerDynArray`.

Благодаря использованию динамических массивов, мы можем получить всю информацию об элементах каталога лишь при помощи трех вызовов удаленного интерфейса. Такой подход сокращает нагрузку не только на сеть, но и на приложение-сервер, так как уменьшает количество загрузок файла catalog.xml. Использование динамических массивов позволяет сэкономить сетевой трафик еще одним способом: получив данные о наименованиях всех элементов каталога, мы можем затребовать остальные данные только о тех элементах, которые нас интересуют (разумеется, необходимо реализовать методы, позволяющие выбирать элементы по определенному критерию).

Примечание

Тот, кто никогда раньше не занимался разработкой SOAP-приложений в средах разработки от Borland, наверное, подумает о том, какие еще методы передачи массивов сложных структур данных можно использовать. Надо сказать, что выбор не так уж велик. Нельзя ли объединить три передаваемых массива в качестве свойств одного класса? Нет, нельзя. Мы можем создать класс-наследник `TRemotable` со свойствами типа "динамический массив", но при создании корректного WSDL-описания такой класс не пройдет. Создать динамический массив классов-наследников `TRemotable` также не получится, так как эти классы обладают свойствами, а шаблон `DynamicArray` не может выполнять операции

с такими классами. При использовании переменной типа "динамический массив записей" компиляция приложения проходит нормально, однако получить WSDL-описание интерфейса, работающего с переменными такого типа не удастся (приложение выдает сообщение "ошибка защиты"). Можно еще, конечно, каким-то образом передавать все данные в одном динамическом массиве, приведя их предварительно к одному типу (например, `String`).

Добавление новых интерфейсов

Не вдаваясь в детали реализации интерфейсов в средах разработки Kylix, приведем пошаговые инструкции по добавлению нового экспортируемого интерфейса в SOAP-сервер на примере языка C++. Для языка Delphi Language все выглядит аналогично.

Добавим к приложению pricelist новый интерфейс `IXMLServ` с единственным методом `getCatalogAsXML`, который возвращает клиенту необработанное содержимое файла catalog.xml. На вкладке **WebServices** диалогового окна **New Items** выберем пункт **SOAP Server Interface**. В открывшемся диалоговом окне укажем имя нового интерфейса (`XMLServ`). После этого будут созданы файлы XMLServ.cpp и XMLServ.h, содержащие заготовки интерфейса, реализующего класс и множество дополнительных подсказок. Для того, чтобы добавить этот интерфейс к уже реализованным на сервере, выполним следующие шаги:

1. Перенесем объявление класса `TXMLServImpl` из файла XMLServ.cpp в файл XMLServ.h.
2. Добавим в раздел `public` интерфейса `IXMLServ` строку

   ```
   virtual String __stdcall getCatalogAsXML() = 0;
   ```

3. Добавим в раздел `public` объявления класса `TXMLServImpl` строку

   ```
   virtual String __stdcall getCatalogAsXML();
   ```

4. Добавим в файл XMLServ.cpp реализацию метода getCatalogAsXML(). Мы можем также удалить из файлов все закомментированные "подсказки".
5. Далее необходимо изменить функцию RegTypes из файла dmList.cpp. В файле XMLServ.cpp есть своя реализация функции RegTypes, но в приложении-сервере может быть определена только одна функция с таким именем. Эта функция вызывается при запуске приложения (обратите внимание на директиву #pragma startup RegTypes 32).
6. Добавим в файл dmList.cpp строку

   ```
   #include "XMLServ.h"
   ```

7. Теперь перенесем текст функции `XMLServFactory` из файла XMLServ.cpp в файл dmList.cpp и изменим текст функции `RegTypes` в файле dmList.cpp на приведенный в листинге 9.11.

Листинг 9.11. Новая функция RegTypes

```
void static RegTypes()
{
  TInvokableClassRegistry * registry = InvRegistry();
  registry->RegisterInterface(__delphirtti(IXMLServ));
  registry->RegisterInvokableClass(__classid(TXMLServImpl),
  XMLServFactory);
  registry->RegisterInterface(__interfaceTypeinfo(IListServ));
  registry->RegisterInvokableClass(__classid(TListServ),
  ListServFactory);
}
```

Функцию `RegTypes` и соответствующую ей директиву `#pragma` в файле XMLServ.cpp следует удалить.

После всех этих операций файлы XMLServ.cpp и XMLServ.h должны соответствовать содержимому листингов 9.12 и 9.13.

Листинг 9.12. Файл XMLServ.h

```
//---------------------------------------------------------------------- //
// Invokable interface declaration header for XMLServ
//---------------------------------------------------------------------- //
#ifndef   __XMLServ_h__
#define   __XMLServ_h__

#include <System.hpp>
#include <InvokeRegistry.hpp>
#include <XSBuiltIns.hpp>
#include <Types.hpp>

__interface INTERFACE_UUID("{0C580F63-F2C7-D611-87DC-00C0DF0894B8}")
IXMLServ : public IInvokable
{
public:
   virtual String __stdcall getCatalogAsXML() = 0;
};
typedef DelphiInterface<IXMLServ> _di_IXMLServ;

class TXMLServImpl : public TInvokableClass, public IXMLServ
{
```

```
public:
  virtual String __stdcall getCatalogAsXML();
  /* IUnknown */
  #ifndef STDMETHODCALLTYPE
     #define STDMETHODCALLTYPE __stdcall
  #endif
  #ifndef S_OK
     #define S_OK 0
  #endif
  #ifndef E_NOINTERFACE
     #define E_NOINTERFACE -1
  #endif

  HRESULT STDMETHODCALLTYPE QueryInterface(const GUID& IID, void **Obj)
                        { return GetInterface(IID, Obj) ? S_OK :↳
  E_NOINTERFACE; }
#ifdef __windows__
  ULONG STDMETHODCALLTYPE AddRef() { return TInterfacedObject::_AddRef();
}
  ULONG STDMETHODCALLTYPE Release() { return
TInterfacedObject::_Release();  }
#endif
#ifdef __linux__
int STDMETHODCALLTYPE AddRef()
{
return TInterfacedObject::_AddRef();  }
int STDMETHODCALLTYPE Release()
{
return TInterfacedObject::_Release();  }
#endif
};

#endif // __XMLServ_h__
//---------------------------------------------------------------------------
```

Листинг 9.13. Файл XMLServ.cpp

```
//********************************************************************* //
// Implementation class for interface IXMLServ
//********************************************************************* //
```

```
#include <stdio.h>
#include <clx.h>
#pragma hdrstop
#include "dmList.h" // Для доступа к константе XML_DOC
#if !defined(__XMLServ_h__)
#include "XMLServ.h"
#endif

String __stdcall TXMLServImpl::getCatalogAsXML()
{
  TStringList *Strings = new TStringList();
  Strings->LoadFromFile(XML_DOC);
  String res = (String)Strings->Text;
  delete Strings;
  return res;
}
```

Теперь приложение pricelist экспортирует и интерфейс `IXMLServ`, в чем можно убедиться, открыв в браузере ссылку **http://localhost/cgi-bin/pricelist** (если мы компилировали приложение как CGI-модуль).

Дополнительные возможности компонента *HTTPRIO*

Мы уже видели, как на основе соответствующего WSDL-описания компонент `HTTPRIO` создает экземпляры экспортируемых SOAP-сервером интерфейсов. Кроме этого, компонент `HTTPRIO` обладает еще рядом полезных возможностей.

Свойство `WSDLItems` компонента `HTTPRIO` позволяет осуществлять разбор XML-документов. Будучи потомком классов `TWSDLDocument` и `TXMLDocument`, класс `TWSDLItems` предоставляет такое свойство, как `Definition`, позволяющее получить интерфейс `IDefinition`, который, в свою очередь, делает доступными все элементы WSDL-документа. С помощью же свойства `DocomentElement` класса `TWSDLItems` можно выполнять разбор WSDL-документа, как любого XML-документа.

При написании приложений-клиентов мы использовали WSDL-файлы, сохраненные на диске как для автоматической генерации описаний экспортируемых интерфейсов, так и для инициализации компонента `HTTPRIO`. При таком подходе нам придется распространять WSDL-файл вместе с нашим приложением, что не очень удобно и в принципе не обязательно.

Напишем SOAP-клиент, обращающийся к интерфейсу IXMLServ на языке Delphi Language. Сохраним WSDL-файл интерфейса IXMLServ на диске и сгенерируем для него описание на Delphi Language так же, как мы это делали выше. Между прочим, из полученного файла IXMLServ1.pas нам необходима только декларация самого интерфейса. Создадим новое приложение в среде Kylix Delphi IDE, добавим в его главную форму компоненты HTTPRIO, Memo и компонент-кнопку Button (назовем соответствующий объект RefreshButton). Перенесем объявление интерфейса IXMLServ из файла IXMLServ1.pas в файл главного модуля приложения клиента. Текст главного модуля приводится в листинге 9.14.

Листинг 9.14. Главный модуль приложения-клиента SOAP для интерфейса IXMLServ

```
unit Main;

interface

uses
  SysUtils, Types, Classes, Variants, QTypes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls, Rio, SOAPHTTPClient;

type

  IXMLServ = interface(IInvokable)
  ['{094AC4EB-6AAE-3864-8478-0794ECCB0807}']
    function  getCatalogAsXML: WideString; stdcall;
  end;

  TForm1 = class(TForm)
    RefreshButton: TButton;
    Memo1: TMemo;
    HTTPRIO1: THTTPRIO;
    procedure FormCreate(Sender: TObject);
    procedure RefreshButtonClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  HTTPRIO1.WSDLLocation :=
'http://first.intranet:8080/cgi-bin/pricelist/wsdl/IXMLServ';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  XS : IXMLServ;
begin
  XS := HTTPRIO1 as IXMLServ;
  Memo1.Text := XS.getCatalogAsXML();
  XS := nil;
end;

end.
```

Как видим, весь код приложения удалось разместить в одном модуле. Более того, сохраненный на диске файл IXMLServ.wsdl нам больше не нужен, так как объект `HTTPRIO1` может считать WSDL-описание интерфейса `IXMLServ` прямо из Интернета во время выполнения программы.

Глава 10

Технология CORBA

В этой главе мы рассмотрим еще одну технологию создания распределенных приложений в рамках архитектуры клиент-сервер. Речь, как можно догадаться, пойдет о технологии CORBA.

Модель CORBA

Аббревиатура CORBA расшифровывается, как Common Object Request Broker Architecture (универсальная архитектура брокеров запросов объектов). CORBA представляет собой спецификацию, разработанную группой OMG (Object Management Group), которая является независимым консорциумом компаний и отраслевых экспертов, вырабатывающих стандарты открытых и кросс-платформенных архитектур распределенных объектов. В отличие от некоторых конкурирующих стандартов (например, таких как технологии COM/DCOM компании Microsoft), группа OMG не предлагает никаких реализаций определяемых ею стандартов.

Ядром архитектуры CORBA является брокер запросов объектов (ORB, Object Request Broker). ORB-брокер является связующим звеном, которое соединяет воедино компоненты распределенной системы. Все транзакции между клиентом и сервером осуществляются через ORB-брокер, причем таким образом, чтобы вызовы методов и их параметры можно было разрешить в адресном пространстве вызывающей или вызываемой процедуры (маршалинг). Брокер ORB также предоставляет много вспомогательных процедур, которые могут быть вызваны прямо из клиента или сервера. Как упоминалось выше, спецификация CORBA не предоставляет никакой стандартной реализации библиотеки ORB. На платформе ОС Linux весьма распространенным является открытый ORB-брокер ORBit, а в состав поставки Kylix входит собственный ORB-брокер VisiBroker.

Интерфейсы CORBA

Распределенная система на основе CORBA может включать разнообразные объекты, реализованные на разных языках программирования и выполняющиеся на разных платформах. Следовательно, как и в случае протокола SOAP, необходим некоторый стандартный способ описания объектов и ме-

тодов взаимодействия с ORB-брокером. Также как и в протоколе SOAP, компоненты распределенной системы экспортируют интерфейсы, описывающие объекты. Как и в протоколе SOAP, за каждым интерфейсом стоит объект, реализующий все методы, экспортируемые данным интерфейсом.

После всего сказанного естественно ожидать, что в CORBA существует свой аналог языка WSDL (Web Services Description Language) — кросс-платформенного языка описания экспортируемых интерфейсов. В рамках технологии CORBA эту роль выполняет язык IDL (Interface Definition Language, язык определения интерфейсов). Язык IDL — это стандартный язык, предназначенный для определения CORBA-интерфейсов.

Заглушки и каркасы

Механизм архитектуры CORBA работает на основе передачи полномочий. В терминах архитектуры CORBA уполномоченный представитель сервера, с которым связывается клиент, называется заглушкой (stub), а представитель клиента на стороне сервера — каркасом (skeleton). ORB-брокеры осуществляют взаимодействие с заглушкой клиента и каркасом сервера, обеспечивая все необходимые операции по передаче данных, включая использование сетевых интерфейсов.

Общая схема модели взаимодействия CORBA-сервера и клиента представлена на рис. 10.1.

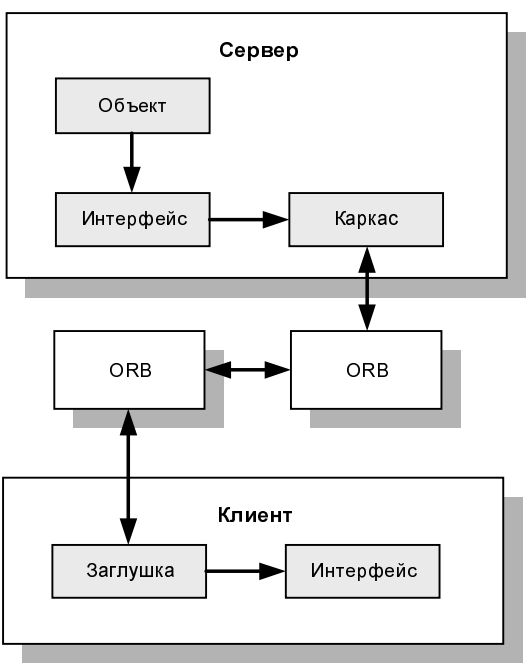


Рис. 10.1. Взаимодействие между CORBA-сервером и клиентом

Менеджер запросов VisiBroker

ORB-брокер VisiBroker поставляется вместе с Kylix 3. Если в процессе установки Kylix мы отметили пункт **VisiBroker/CORBA support**, то приложение VisiBroker должно быть установлено в каталоге kylix3/vbroker. Прежде чем запускать сам ORB-брокер, необходимо выполнить скрипт `vbroker.sh`, находящийся в этом каталоге. Скрипт `vbroker.sh` установит необходимые для работы ORB-брокера значения переменных окружения `OSAgent` и `LD_LIBRARY_PATH`. После этого можно запустить и сам ORB-брокер командой:

```
./osagent
```

Мы можем также запустить ORB-брокер VisiBroker в фоновом режиме. Для этого следует воспользоваться командой:

```
./osagent &
```

При запуске в фоновом режиме ORB-брокер распечатает в окне терминала значение своего идентификатора `pid`. Это значение можно затем использовать для остановки ORB-брокера командой `kill`.

Разработка клиентов и серверов CORBA

Средства разработки Borland позволяют генерировать модули, содержащие объявления интерфейсов, на основе описаний этих интерфейсов на языке IDL. В состав поставки Kylix 3 входит специальный компилятор idl2pas, генерирующий необходимые модули Delphi Language на основе данных, содержащихся в IDL-файле. Компилятор idl2.pas интегрирован в Kylix IDE, так что необязательно вызывать его непосредственно.

Напишем простые CORBA-клиент и CORBA-сервер. Наш CORBA-сервер экспортирует интерфейс `ICurrentTime`, включающий единственный метод `GetTime`, возвращающий строку, содержащую текущее значение времени.

В соответствии с моделью разработки, принятой в среде Kylix, нам сначала придется написать IDL-файл, описывающий интерфейсы. Правила написания IDL-файлов достаточно просты, и их легко понять из приводимого примера (листинг 10.1).

Листинг 10.1. Файл CurrentTime.idl

```
module CurrentTime
{
  interface ICurrentTime;

  interface ICurrentTime
  {
    string GetTime();
```

```
  };

  interface CurrentTimeFactory
  {
    ICurrentTime CreateInstance(in string InstanceName);
  };
};
```

Как видим, синтаксис описания интерфейсов очень похож на синтаксис языка C++. Интерфейс `CurrentTimeFactory` необходим для автоматического создания копии соответствующего объекта при обращении к экспортируемому интерфейсу `ICurrentTime`.

Теперь мы можем приступить к написанию приложения-сервера.

1. Выбираем пункт **CORBA Application Server** на странице **CORBA** диалогового окна **New Items**. Открывается окно **IDL2Pas Create Server Dialog** (рис. 10.2), в котором можно указать тип приложения (консольное или графическое) и добавить IDL-файл (кнопка **Add**).
2. Добавляем в проект файл CurrentTime.idl.

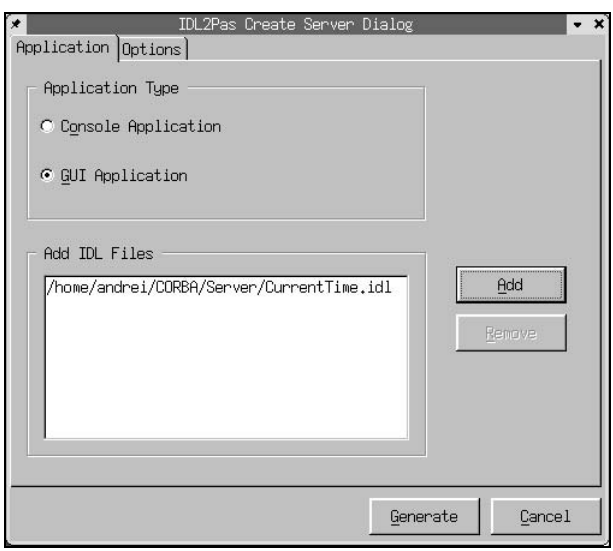


Рис. 10.2. Окно **IDL2Pas Create Server Dialog**

На вкладке **Options** задаются дополнительные параметры для генерации исходных текстов. Проверим, чтобы на этой вкладке были установлены флажки **Generate Skeleton Units** и **Generate Implementation Units**, которые соответственно указывают на необходимость генерации модулей, содержащих каркасы, и модулей, содержащих реализации интерфейсов. После этого нажимаем кнопку **Generate**.

В результате будут созданы четыре файла на Delphi Language со следующими именами: CurrentTime_i.pas, CurrentTime_s.pas, CurrentTime_c.pas и CurrentTime_impl.pas. Файл CurrentTime_i.pas содержит реализацию интерфейса `ICurrentTime`, файл CurrentTime_s.pas содержит реализацию класса-каркаса `TICurrentTimeSceleton`. Файл CurrentTime_c.pas содержит класс, реализующий заглушку, а также некоторые вспомогательные классы, а файл CurrentTime_impl.pas содержит реализацию функций экспортируемого интерфейса.

В файл CurrentTime_impl.pas следует добавить код, реализующий метод `GetTime` (листинг 10.2).

Листинг 10.2. Реализация метода `GetTime`

```
function TICurrentTime.GetTime : String;
begin
  Result := TimeToStr(Time);
end;
```

Используемые в этом методе функции объявлены в модуле `SysUtils`, который следует включить в раздел `uses` файла CurrentTime_impl.pas.

В текст главного модуля приложения-сервера необходимо внести некоторые изменения, чтобы он соответствовал тексту, приведенному в листинге 10.3.

Листинг 10.3. Главный модуль приложения-сервера

```
unit Main;

interface

uses
  {$ifdef MSWINDOWS}
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Corba, CurrentTime_I, CurrentTime_C, CurrentTime_S,
  CurrentTime_Impl, StdCtrls;
  {$endif}
  {$ifdef LINUX}
  SysUtils, Classes, QGraphics, QControls, QForms, QDialogs, CORBA,
  CurrentTime_i, CurrentTime_c, CurrentTime_s, CurrentTime_impl,
  QStdCtrls;
  {$endif}

type
  TForm1 = class(TForm)
```

```
  procedure FormCreate(Sender: TObject);
  private
  { private declarations }
  protected
  { protected declarations }
    GT : ICurrentTime; // skeleton object
    procedure InitCorba;
  public
  { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.InitCorba;
begin
  CorbaInitialize;

  // Add CORBA server Code Here
  GT := TICurrentTimeSkeleton.Create('Time Server',
  TICurrentTime.Create);
  BOA.ObjIsReady(GT as _Object);
end;

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

Теперь приложение-сервер готово. Мы можем запустить его (запустив предварительно VisiBroker).

Нам следует создать еще приложение-клиент. Выбираем пункт **CORBA Aplication Client** на той же странице **CORBA** диалогового окна **New Items**.

1. В открывшемся окне **IDL2Pas Create Client Dialog** добавляем файл CurrentTime.idl. Вместе с заготовкой приложения клиента создаются файлы CurrentTime_i.pas и CurrentTime_c.pas, но мы можем воспользоваться ана-

логами этих файлов, созданными в процессе разработки приложения-сервера.

2. Добавим в главную форму приложения-клиента два компонента `Label` и кнопку `Button` (назовем ее `GetTimeButton`).
3. Кроме того, в разделе `protected` класса `TForm1` следует ввести переменную `CurrentTime` типа `ICurrentTime`.

Полный исходный текст главного модуля приложения-клиента приводится в листинге 10.4.

Листинг 10.4. Главный модуль приложения-клиента

```
unit ClientMain;

interface

uses
  {$ifdef MSWINDOWS}
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Corba, StdCtrls, CurrentTime_I, CurrentTime_C;
  {$endif}

  {$ifdef LINUX}
  SysUtils, Classes, QGraphics, QControls, QForms, QDialogs,
  CORBA, QStdCtrls, CurrentTime_i, CurrentTime_c;
  {$endif}

type
  TForm1 = class(TForm)
    GetTimeButton: TButton;
    Label1: TLabel;
    Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure GetTimeButtonClick(Sender: TObject);
  private
  { private declarations }
  protected
  { protected declarations }
    CurrentTime : ICurrentTime;
   procedure InitCorba;
  public
  { public declarations }
```

```
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.InitCorba;
begin
  CorbaInitialize;
  CurrentTime := TICurrentTimeHelper.bind;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 Label1.Caption := 'Текущее время';
 Label2.Caption := '';
  GetTimeButton.Caption := 'Обновить';
 InitCorba;
 end;

procedure TForm1. GetTimeButtonClick(Sender: TObject);
begin
  Label2.Caption := CurrentTime.GetTime;
end;
end.
```

Приложение-клиент инициализирует экспортируемый интерфейс при помощи метода

`TICurrentTimeHelper.bind`.

Теперь можно запустить и клиент и сервер (не забыв запустить предварительно брокер VisiBroker). При нажатии на кнопку **Обновить** приложение-клиент должно вывести значение текущего времени (рис. 10.3).

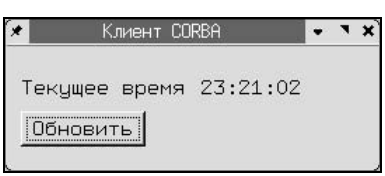


Рис. 10.3. Приложение-клиент CORBA

Часть III

Работа с базами данных

Глава 11

Принципы разработки приложений баз данных в Kylix 3

Часть III этой книги посвящена программированию приложений для работы с базами данных. Мы рассмотрим не только локальные, но и распределенные приложения баз данных, так что эту часть книги нельзя рассматривать как полностью независимую от предыдущих. Напротив, в этой части мы снова встретимся с технологиями WebSnap и WebBroker, так что, если при чтении книги главы, посвященные этим технологиям, были пропущены, то рекомендуем их прочитать.

Как всегда, описание практической разработки приложений в среде Kylix 3 будет предваряться небольшим теоретическим введением. Поэтому данная глава посвящена теории самих баз данных и общему описанию структуры и принципов построения приложений баз данных в Kylix 3.

При написании этой части ставилась задача, чтобы изложение каждой темы сопровождалось иллюстрированными примерами. Поскольку примеры, как правило, представляют собой законченные приложения, при таком способе изложения неизбежно возникает некоторое "забегание вперед", когда в примерах приходится использовать элементы, которые еще подробно не освещались. Однако такой способ изложения все-таки лучше, чем строго последовательный, при котором средства и методы разработки приложений приходится описывать абстрактно, так как для создания работающего примера еще попросту не хватает материала. Именно этой принятой здесь концепцией изложения и обусловлен порядок следующих глав. Поскольку любой пример приложения баз данных нуждается прежде всего в базе данных, главы 12 и 13 посвящены описанию работы двух распространенных систем управления базами данных (СУБД) и созданию простейших приложений для них с использованием элементов, подробно описанных в последующих главах. В главе 14 мы познакомимся с основами языка SQL (Structured Query Language, язык структурированных запросов) и компонентами dbExpress, позволяющими устанавливать связь между приложением и базами

данных. Глава 15 посвящена локальным приложениям баз данных, при этом подробно будут рассматриваться клиентские наборы данных и элементы графического интерфейса приложений баз данных — темы, которые в соответствии с принятой концепцией уже были бегло рассмотрены в главах, предшествующих главе 15. Наконец, глава 16 — единственная глава этой части, в которой не придется "забегать вперед", будет посвящена распределенным приложениям баз данных.

Реляционная модель баз данных

Практически все базы данных, с которыми мы будем иметь дело в этой книге, являются реляционными. Поэтому естественно будет начать с краткого описания реляционной модели баз данных.

Реляционная модель описывает структуру данных, которые могут храниться в реляционных базах данных, а также способы манипулирования этими данными. В упрощенном виде основная идея реляционной модели состоит в том, что данные должны храниться в таблицах и только в таблицах.

Классическая реляционная модель данных требует, чтобы данные хранились в так называемых *плоских таблицах*. Точнее сказать, пользователи и приложения, обращающиеся к данным, должны работать с данными так, как если бы они размещались в таких таблицах. В упрощенном виде плоскую таблицу можно рассматривать как таблицу, каждая ячейка которой может быть однозначно идентифицирована номером строки и столбца. Кроме того, в одном столбце все ячейки должны содержать данные одного простого типа.

Реляционная модель баз данных состоит из трех частей:

- структурной;
- целостной;
- манипуляционной.

Из этих трех частей мы подробно рассмотрим первую, структурную часть, которая тесно связана со многими понятиями и определениями языка запросов SQL. Этот язык использован в СУБД, которые будут рассмотрены нами в последующих главах этой части книги.

Структурная часть описывает объекты, входящие в реляционную модель баз данных. Как уже говорилось, в рамках реляционной модели таблицы должны содержать только данные простых типов. Простые, или атомарные, типы данных не обладают внутренней структурой. Данные такого типа называют *скалярами*. К простым типам данных относятся следующие типы:

- логический;
- строковый;
- численный.

На практике этот список простых типов дополняется следующим:

- целый;
- вещественный;
- дата;
- время;
- денежный;
- перечислимый.

Собственно говоря, для реляционной модели данных тип используемых данных не важен. Требование, чтобы тип данных был простым, фактически означает, что в реляционных операциях с данными не должна учитываться внутренняя структура данных. Таким образом, в реляционной модели данных каждый тип данных рассматривается как единое целое. Разумеется, для каждого такого типа данных можно ввести специальные определения операций, например, операции сравнения.

В реляционной модели данных с понятием типа данных тесно связано понятие *домена*, которое можно считать уточнением типа данных.

Домен можно рассматривать как подмножество значений некоторого типа данных, имеющих определенный смысл. Домен характеризуется следующими свойствами:

- уникальное имя (в пределах базы данных);
- домен определяется на некотором простом типе данных или на другом домене;
- домен может иметь некоторое логическое условие, позволяющее описать подмножество данных, допустимых для данного домена;
- обычно домен несет определенную смысловую нагрузку.

Например, домен, имеющий смысл "зарплата сотрудника" можно описать как следующее подмножество множества натуральных значений.

Если тип данных можно считать множеством всех возможных значений данного типа, то домен напоминает подмножество в этом множестве. Отличие домена от понятия подмножества состоит именно в том, что домен отражает не только содержание, но и смысл определенной области данных. Может быть несколько доменов, совпадающих как подмножества, но несущих различный смысл. Например, домены "зарплата сотрудника" и "возраст сотрудника" можно одинаково описать как множество неотрицательных целых чисел, но смысл этих подмножеств будет различным и они будут представлены различными доменами.

Основное значение доменов состоит в наложении смысловых ограничений на операции сравнения. Некорректно сравнивать значения из различных доменов, даже если они имеют одинаковый тип. В этом проявляется смы-

словое ограничение доменов. Синтаксически правильный запрос "выдать список всех сотрудников, у которых размер зарплаты превышает возраст" не соответствует смыслу понятий "размер зарплаты" и "возраст".

Одним из основных преимуществ реляционного подхода к организации баз данных является то, что пользователи реляционных БД получают возможность эффективной работы с использованием простых и наглядных понятий таблиц, их строк и столбцов. При этом пользователям не нужно знать детали реальной организации данных во внешней памяти. Реляционная модель данных позволяет пользователям работать в ненавигационной манере, т. е. для выборки информации из БД клиент должен всего лишь указать список интересующих его таблиц и те условия, которым должны удовлетворять выбираемые данные. СУБД скрывает от пользователя выполняемые ею последовательные просмотры таблиц и алгоритмы, при помощи которых выполняются эти операции. Основная особенность реляционных систем состоит в том, что результатом выполнения любого запроса к таблицам баз данных также является таблица, которую можно отображать в клиентском приложении, сохранять в других БД или выполнять по отношению к собственной БД новые запросы.

Для эффективной работы реляционные СУБД нуждаются в мощном и в тоже время простом языке, позволяющем выполнять все необходимые пользователям операции. В последние годы таким повсеместно принятым языком стал язык SQL, более подробное описание которого приводится в главе 14.

Понятие транзакции

Прежде, чем приступить к описанию архитектур систем управления базами данных, необходимо дать точное определение термину *транзакция*, играющему в описании работы баз данных важную роль.

Примечание

Термин *транзакция* (transaction) получился в результате слияния слов из английского словосочетания transformation action, что можно перевести как "преобразующее действие".

Транзакция — это операция или группа операций, преобразующих базу данных из одного целостного состояния в другое. Транзакция характеризуется четырьмя основными свойствами.

- Атомарность (atomicity) — означает, что транзакция либо выполняется целиком, либо не выполняется вообще. Если выполнение транзакции было прервано до ее корректного завершения, система возвращается в состояние, предшествовавшее состоянию выполнения транзакции.

- Целостность (consistency) — транзакция должна переводить базу данных из одного целостного состояния в другое. Это означает, что транзакция не должна нарушать структуру данных, определенную бизнес-правилами (логикой базы данных).
- Изолированность (isolation) — транзакция не должна взаимодействовать с другими транзакциями, выполняющимися на сервере одновременно с данной транзакцией.
- Действенность (durability) — изменения, внесенные транзакцией, должны вступить в силу после получения специального подтверждения, которое означает, что транзакция выполнена.

При всей важности механизма транзакций в современных СУБД его не следует считать универсальным. Например, в СУБД MySQL, кроме механизма транзакций, реализован еще один механизм поддержания целостности при операциях с данными, называемый Atomic Operations. Разработчики MySQL считают этот механизм более совершенным, чем механизм транзакций.

Архитектура СУБД

Системы управления базами данных (СУБД) подразделяются на несколько категорий, каждая из которых будет кратко описана ниже.

Локальная архитектура

В рамках этого типа архитектуры и программа, и база данных расположены на одном компьютере. Такая архитектура характерна для большинства настольных (desktop) приложений.

Файл-серверная архитектура

В рамках этой архитектуры база данных (файлы, в которых физически хранятся данные) расположена на отдельном компьютере (сервере), к которому посредством сети подключены компьютеры-клиенты. На этих компьютерах установлены клиентские программы, обращающиеся к базе данных по сети. Преимущество такой архитектуры заключается в больших возможностях параллельной работы с одной базой данных.

Необходимо подчеркнуть, что отличительной особенностью файл-серверной архитектуры является то, что сервер выступает лишь в роли хранилища данных, а вся обработка данных выполняется на стороне клиентов. С этим связан и основной недостаток этой архитектуры — большие объемы информации, передаваемой по сети. Это приводит к ограничению максимально возможного числа пользователей и большим задержкам при работе с базой данных. Задержки вызываются тем, что на

уровне конкретной таблицы одновременный доступ не возможен. Пока один из клиентов не завершит транзакцию, связанную с данной таблицей, другие программы не могут обращаться к этой таблице. Такой механизм носит название *блокировки на уровне таблицы* и обеспечивает сохранение целостности данных.

Клиент-серверная архитектура

При такой архитектуре на сервере располагается не только база данных, но и программа СУБД, обрабатывающая запросы пользователей и возвращающая им наборы записей. При этом программы-клиенты уже не работают напрямую с базой данных как с набором физических файлов, а обращаются к СУБД, которая выполняет требуемые операции. Нагрузка на программы-клиенты при этом существенно уменьшается, так как большая часть операций выполняется на сервере. СУБД автоматически следит за целостностью и сохранностью базы данных, а также контролирует доступ к информации с помощью системы разграничения прав доступа. Клиент-серверные СУБД допускают блокировку на уровне записи и даже отдельного поля. Это означает, что с каждой данной таблицей одновременно может работать любое число пользователей, но доступ к функции изменения конкретной записи или одного из ее полей в каждый данный момент времени предоставлен только одному из них. Основной недостаток этой архитектуры — не очень высокая надежность. Выход сервера из строя парализует работу всей системы.

Распределенная архитектура

При реализации этой архитектуры в сети работает несколько серверов, и таблицы баз данных распределены между ними для повышения эффективности работы. На каждом сервере функционирует своя копия СУБД. Кроме того, в подобной архитектуре обычно используются специальные программы так называемые — серверы приложений. Они позволяют оптимизировать обработку запросов большого числа пользователей и равномерно распределить нагрузку между компьютерами в сети. Если, помимо работы с данными, требуется выполнить интенсивные вычисления (например, анализ сложной информации), программы для выполнения этих задач могут автоматически запускаться на менее нагруженных сетевых компьютерах. В рамках этой архитектуры нагрузка на программы-клиенты также крайне невелика. Недостаток распределенной архитектуры заключается в довольно сложном и дорогостоящем процессе ее создания и администрирования, а также в высоких требованиях, предъявляемых к серверным компьютерам.

Интернет-архитектура

Доступ к базе данных и СУБД (которые могут быть расположены как на одном компьютере, так и на разных, связанных между собой сетью) осуществляется посредством интернет-браузера по стандартному протоколу. Требования, предъявляемые в рамках этой архитектуры к клиентскому оборудованию, минимальны. Клиентов такой базы данных называют "тонкими клиентами", потому что они используют стандартные протоколы и интерфейсы, а также типовые линии связи. В результате система получается недорогой и простой в установке и эксплуатации. Естественно, данная архитектура является оптимальной при создании базы данных, доступной в Глобальной сети, где пользователи должны иметь возможность получать доступ к базе данных посредством стандартных клиентских сетевых программ.

Данная архитектура разделяет все недостатки, присущие работе в глобальных сетях (низкая пропускная способность каналов связи, сравнительно низкая надежность связи и т. п.).

Структура приложений баз данных в Kylix 3

В процессе разработки приложения баз данных создается цепочка компонентов, связывающая источник данных, с одной стороны, и пользовательский интерфейс приложения, с другой стороны. Общая структура приложения баз данных показана на рис. 11.1.

На первый взгляд, цепочка компонентов, связывающих базу данных с интерфейсом пользователя, может показаться слишком длинной и сложной, но на самом деле наличие каждого элемента в этой цепочке вполне оправдано. Мы видели, что приложения, работающие с базами данных, могут иметь самую разную архитектуру, начиная с локальной и заканчивая распределенной архитектурой в Глобальной сети.

Наличие большого числа элементов в цепочке компонентов делает модель приложения для работы с базами данных чрезвычайно гибкой. С помощью этой модели можно создавать приложения для взаимодействия с СУБД всех вышеперечисленных архитектур. Кроме того, поскольку отдельные звенья этой цепочки абстрагированы друг от друга и взаимодействуют на основе общих интерфейсов, каждое звено цепочки может быть заменено другим звеном того же типа. Таким образом, приложение для работы с базами данных одного типа может быть легко преобразовано в приложение для работы с базами данных другого типа. Следует отметить, что для приложений, работающих с SQL-базами данных, описанная ниже цепочка компонентов может быть существенно упрощена за счет использования компонента `SQLClientDataSet`, который инкапсулирует несколько звеньев этой цепочки *(подробную информацию см. в гл. 14)*.

Из приведенной схемы видно, что приложение для работы с базами данных состоит из трех частей:

- интерфейса пользователя;
- компонентов доступа к базам данных;
- базы данных.

К компонентам пользовательского интерфейса относятся компоненты, расположенные на странице **Data Controls** палитры компонентов. Многие компоненты страницы **Data Controls** являются аналогами компонентов пользовательского интерфейса. Отличие компонентов пользовательского интерфейса приложений баз данных заключается в том, что эти компоненты динамически связаны с источниками данных. К этой же категории следует отнести компоненты `DataSetTableProducer` и `DataSetPageProducer`, расположенные на странице **Internet** палитры компонентов.

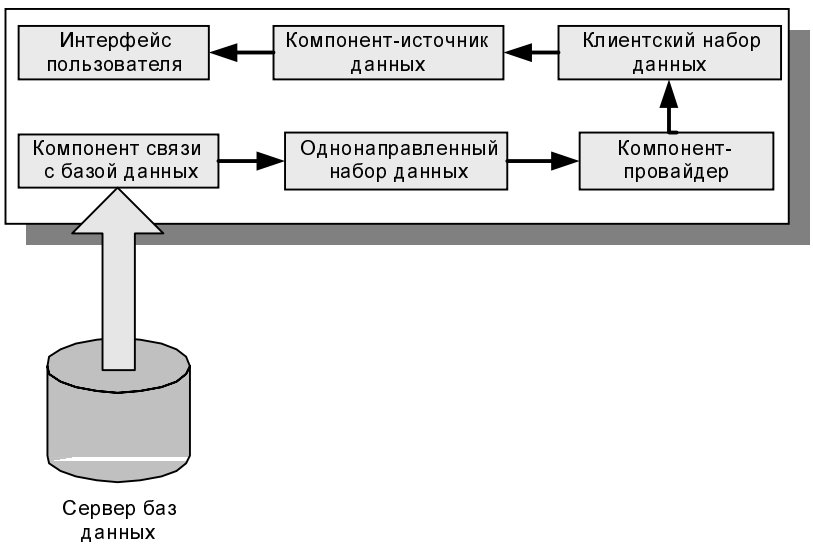


Рис. 11.1. Структура приложения баз данных в Kylix 3

Компонент-источник данных играет роль "крючка", к которому прикрепляются различные элементы пользовательского интерфейса.

Компоненты доступа к базам данных расположены на страницах **Data Access** и **DbExpress**. Компоненты панели **Data Access** включают в себя компонент, реализующий клиентский набор данных (компонент `ClientDataSet`), компонент-провайдер (`DataSetProvider`) и ряд компонентов, предназначенных для взаимодействия приложений баз данных с XML-документами.

Основное различие между однонаправленным и клиентским наборами данных заключается в том, что клиентский набор данных хранит в памяти мас-

сив записей, полученных в результате запроса к серверу баз данных. Компоненты, реализующие клиентские наборы данных, позволяют также редактировать записи и вносить изменения в базу данных. Однонаправленные наборы данных могут работать в каждый момент времени лишь с одной записью, при этом произвольный доступ к записям, поддерживаемый клиентскими наборами данных, невозможен.

Компоненты, реализующие клиентские наборы данных, получают данные от компонентов-провайдеров, которые формируют *пакеты данных* на основе блоков данных, полученных от однонаправленного набора данных или другого источника.

В состав компонентов панели **dbExpress**, в числе других, входят компонент `SQLDataSet`, реализующий однонаправленный набор данных, и компонент `SQLConnection`, позволяющий устанавливать связь с SQL-серверами баз данных. Кроме того, в набор компонентов панели **dbExpress** входит и упоминавшийся выше компонент `SQLClientDataSet`.

Для того чтобы клиентский набор данных мог передать информацию компоненту-источнику данных для отображения в графических компонентах, он должен быть открыт. Открытый клиентский набор данных посылает запрос компоненту-провайдеру на формирование пакета данных. Компонент-провайдер открывает однонаправленный набор данных и последовательно считывает записи из таблицы, к которой однонаправленный набор данных получает доступ с помощью компонента, осуществляющего связь с базой данных. Из этих записей компонент-провайдер формирует пакет данных, который передается компоненту, реализующему клиентский набор данных.

Если пользователь вносит изменения в базу данных с помощью графического элемента управления, измененные записи сначала передаются клиентскому набору данных, который передает их компоненту-провайдеру, который, в свою очередь, передает их с помощью соответствующих методов однонаправленного набора данных компоненту, осуществляющему связь с базой данных.

Компоненты набора Data Access `XMLTransformClient` и `XMLTransformProvider`, позволяют использовать в качестве хранилища данных XML-документы, и работать с ними в рамках стандартной модели приложений баз данных. Кроме того, эти компоненты позволяют организовать обмен информацией между XML-файлами и базами данных.

Для компонентов, позволяющих изменять данные, необходимы: явный компонент-провайдер и клиентский набор данных, позволяющие обновлять содержимое базы данных.

Компонент `SQLConnection` является центральным звеном приложений, работающих с SQL-СУБД. Этот компонент инкапсулирует несколько классов dbExpress, предназначенных для непосредственного доступа к различным ба-

зам данных. Для каждой поддерживаемой базы данных dbExpress предоставляет драйвер, который позволяет другим компонентам dbExpress обращаться к базам данных, используя единый интерфейс компонента `SQLConnection`. В настоящий момент dbExpress в Kylix Enterprise Edition поддерживает следующие базы данных: IBM DB2, Borland InterBase, Informix, MySQL, PostgreSQL, Oracle.

При распространении приложения, предназначенного для работы с определенной базой данных, вместе с приложением необходимо распространять соответствующий драйвер dbExpress, реализованный в виде разделяемой библиотеки. Несколько однонаправленных наборов данных (компонентов `SQLClientDataSet`) могут использовать один и тот же компонент `SQLConnection`.

Если эта схема распределения компонентов кажется немного запутанной, а функции отдельных компонентов — дублирующими друг друга, то стоит напомнить, что разработчики Kylix стремились обеспечить максимальную совместимость с Delphi и Borland C++ Builder. В этих же средствах разработки, предназначенных для ОС Windows, набор компонентов для создания приложений баз данных существенно расширен и включает в себя весьма разнородные компоненты доступа к базам данных, основанные не только на языках SQL и XML. Кроме того, следует иметь в виду, что структура компонентов для работы с базами данных проектировалась таким образом, чтобы одни и те же компоненты можно было использовать как в локальных, так и в распределенных приложениях баз данных. Именно этим объясняется в частности то, что компонент-клиентский набор данных не содержит свойства-ссылки на объект-провайдер, а требует указывать имя провайдера в формате строки.

Структура компонентов для работы с базами данных в Kylix организована таким образом, чтобы максимально облегчить перенос приложений между разными платформами.

Глава 12

Работа с СУБД InterBase

СУБД InterBase разрабатывается компанией Borland и поставляется вместе со всеми интегрированными средами разработки этой компании. Так же, как и другие современные СУБД, эта СУБД основана на реляционных базах данных SQL, но предоставляет некоторые специфические расширения.

Примечание

На сегодняшний день СУБД InterBase является коммерческим продуктом, и, хотя Borland обещает сделать свою СУБД открытой и бесплатной, на момент написания этой книги данный проект еще не был реализован. В этом разделе мы описываем коммерческую версию InterBase 6.5.

Естественно, что СУБД InterBase тесно связана со средствами разработки от Borland. В Delphi для ОС Windows есть даже специальный набор компонентов для работы с СУБД InterBase. В Kylix такого набора нет, и работа с СУБД InterBase выполняется на основе тех же компонентов, что и работа с другими поддерживаемыми Kylix СУБД.

Установка и настройка СУБД InterBase в операционной системе Linux

Обычно СУБД InterBase устанавливается в каталог **/opt/interbase/**. Запустить сервер InterBase можно при помощи команды:

```
[root]#  /opt/interbase/bin/ibmgr -start
```

Если сервер запущен успешно, на экране должна появиться строка:

```
server has been successfully started
```

Однако запустить сервер сразу после установки удается не всегда. Часто система нуждается в дополнительной конфигурации для запуска СУБД InterBase. Рассмотрим две наиболее распространенные проблемы и методы их решения.

Если при запуске сервера программа сообщает:

```
address and port are already in use
```

необходимо выполнить следующие действия. Добавим в файл /etc/hosts.equiv строки:

```
+
your_host_name
localhost
```

Здесь `your_host_name` — имя локального узла (сетевое имя нашей машины). В файле /etc/inetd.conf (если таковой существует) добавим символ комментария "#" перед строкой, начинающейся с "gds_db". После этого перезапустим демон `inetd` (самый простой способ сделать это — перезапустить саму операционную систему).

Если в ответ на команду запуска сервер выдает сообщение:

```
cannot connect to the specified host host_name
```

необходимо прежде всего проверить, внесено ли указанное имя узла в файл /etc/hosts, и если нет, внести его туда. Кроме того, необходимо, чтобы в файле /etc/services присутствовала строка:

```
gds_db      3050/tcp
```

После внесения изменений в указанные выше файлы демон `inetd` также необходимо перезапустить.

Создание новых учетных записей в InterBase

Учетные записи пользователей хранятся в специальной базе данных isc4.gdb. Для управления этой базой данных служит утилита gsec. Запустим утилиту gsec командой:

```
/opt/interbase/bin/gsec
```

На экране терминала появится приглашение

```
GSEC>
```

Мы можем просмотреть список существующих учетных записей при помощи команды:

```
GSEC> display
```

Запустив эту утилиту в первый раз, мы увидим данные об учетной записи пользователя SYSDBA. Добавить новую учетную запись можно с помощью команды `add`. Добавим учетную запись пользователя `andy` с паролем `letmein`.

```
GSEC> add andy -pw letmein
```

Команда `add` позволяет вносить в базу данных учетных записей не только учетное имя пользователя и пароль, но также (необязательные) идентификатор пользователя и группы, а также полное имя пользователя. Для выхода из утилиты gsec служит команда `quit`.

Создание баз данных в InterBase и разделение прав пользователей

В составе СУБД InterBase поставляется не только серверная, но и клиентская часть. Роль клиентской части в СУБД InterBase выполняет утилита isql (Interactive SQL, интерактивный SQL). Эта утилита также установлена в каталоге **/opt/interbase/bin/**. Запустим утилиту isql со следующими параметрами:

```
isql -u sysdba -p masterkey
```

Ключ –u позволяет указать имя пользователя, а ключ –p — пароль. Кроме этого в качестве параметров запуска утилиты isql можно указывать и другие ключи, например, ключ –o позволяет записывать данные, выводимые программой в специальный файл, а не на экран терминала.

После этого программа предложит создать новую базу данных или связаться с уже существующей. Создадим новую базу данных:

```
SQL> create database "/home/andrei/DB/catalog.gdb";
```

В отличие от многих других СУБД, все элементы отдельной базы данных InterBase физически хранятся в одном файле, по умолчанию имеющем расширение .gdb.

Большая часть команд isql (которые, в основном, являются обычными командами SQL) должна заканчиваться символом ";". Если мы забудем ввести этот символ или как-то иначе не завершим ввод команды и нажмем клавишу **Ввод**, приглашение программы сменит вид с `SQL` на `CON`. Это означает, что мы должны продолжить ввод команды.

После создания базы данных можно создать таблицу:

```
SQL> create table pricelist (Name VARCHAR(32), Section VARCHAR(10)↳
Manufacturer VARCHAR(15), Price INT, Discount INT);
```

Теперь выйдем из утилиты isql с помощью команды `quit` и войдем снова, на этот раз с параметрами:

```
isql -u andy -p letmein
```

и свяжемся с созданной ранее базой данных:

```
SQL> connect /home/andrei/DB/catalog.gdb
```

Создадим в базе данных новую таблицу:

```
SQL> create table andytable (Name VARCHAR(32), Id INT);
```

Все проходит успешно. Команда `show tables` выводит на экран обе таблицы (`pricelist` и `andytable`), но запрос к таблице `pricelist`:

```
select * from pricelist;
```

приводит к выводу сообщения о нарушении прав доступа. Пользователь andy может получить доступ к таблице catalog.gdb и создавать в ней табли-

цы, однако он не может получить доступ к таблицам, созданным другими пользователями. Таким образом, разграничение доступа в СУБД InterBase осуществляется на основе отдельных таблиц, а не на основе баз данных, к которым каждый пользователь имеет доступ.

Заполнять таблицы InterBase можно, перенося в них данные из текстовых файлов. В СУБД InterBase это можно сделать при помощи команды `EXTERNAL FILE`, например:

```
SQL> create table mytable external file "myfile" (column1 VARCHAR(32),↳
column2 int, ...);
```

Другие варианты использования параметра `EXTERNAL FILE` включают возможность сохранять данные из базы данных InterBase в текстовом файле и работать с текстовым файлом как с базой данных InterBase.

Разработка приложений для СУБД InterBase

Для того чтобы создаваемое нами приложение могло взаимодействовать с СУБД InterBase, необходимо настроить компонент `TSQLConnection`. Дважды щелкнем кнопкой мыши на пиктограмме компонента в окне формы. В открывшемся окне (рис. 12.1) в списке **Connection Name** выберем пункт **IBConnection**. Самые важные поля в таблице **Connection Settings — Database**, **User_Name** и **Password**. Поле **Database** позволяет указать файл базы данных (и, если необходимо, полный путь к нему). Поля **User_Name** и **Password** позволяют задать, соответственно, имя и пароль пользователя. Необходимо помнить, что данный пользователь может не иметь доступа ко всем таблицам базы данных, с которой он работает. Еще одно поле таблицы **Connection Settings — SQLDialect** позволяет указать версию языка SQL, которая используется для формирования запросов серверу. По умолчанию этому полю присваивается значение 1, но лучше выбрать версию 3, поддерживаемую СУБД InterBase, начиная с версии 6.0, так как этот вариант языка SQL обладает более широкими возможностями.

Проверить правильность настройки можно, присвоив значение `true` свойству `Connected` компонента `SQLConnection`. При этом на экране появится окно с запросом на ввод пароля для указанного пользователя. Если после ввода пароля не появится диалоговое окно с предупреждением о возникновении исключительной ситуации, значит, соединение установлено успешно.

Этот пример демонстрирует тот факт, что компоненты приложения баз данных взаимодействуют с сервером баз данных еще на этапе разработки приложения. Этот факт можно использовать для отладки приложений баз данных.

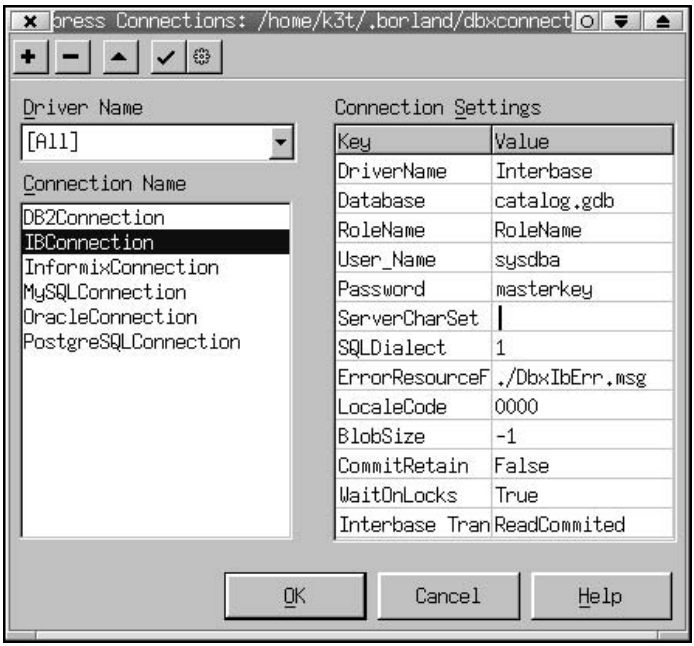


Рис. 12.1. Настройка параметров соединения для связи с СУБД InterBase

Создадим специальную базу данных для приложения-примера. Пусть наша база данных содержит информацию о сотрудниках некоторого предприятия. Самый простой способ заполнить базу данных — импортировать данные из текстового файла. СУБД InterBase предъявляет к таким файлам довольно специфические требования:

- файлы должны состоять из строк одинаковой длины;
- значение каждого поля таблицы должно занимать столько символов, сколько выделено под это значение при объявлении типов полей таблицы, незанятые символы заполняются пробелами;
- отдельные поля в строках файла никак не разделяются, программа определяет, где заканчивается одно поле и начинается другое, по количеству символов от начала строки;
- при подсчете числа символов учитывается символ перехода на новую строку (при этом следует принимать во внимание, что в ОС Linux переход на новую строку кодируется одним ASCII-символом).

Итак, пусть наша таблица содержит четыре столбца Name (имя сотрудника, 10 символов), MiddleName (отчество сотрудника, 15 символов); Surname (фамилия сотрудника, 15 символов) и BirthDate (дата рождения, 9 символов — 8 символов на значение даты, 1 символ — переход на новую строку).

Создадим файл employees.txt, содержание которого приводится в листинге 12.1.

Листинг 12.1. Файл employees.txt

```
Павел      Сергеевич      Королев        01/07/61
Степан     Николаевич     Стрельцов      21/03/68
Андрей     Геннадьевич    Бабушкин       05/04/75
Людмила    Борисовна      Скворцова      13/09/72
```

При заполнении файла нужно следить за тем, чтобы число пробелов дополняло каждое поле до нужного числа символов. Это очень удобно делать, используя фиксированный шрифт.

Войдем в программу isql под именем созданного нами пользователя и введем команды:

```
SQL> create database '/home/username/employees.gdb';
SQL> create table employees (Name char(10), MiddleNaMe char(15),↳
  Surname char(15), BirthDate char(9));
```

Мы создали базу данных employees.gdb с таблицей employees. Теперь создадим проект графического приложения Kylix и разместим в его форме компонент `SQLConnection`. Отредактируем свойства соединения `IBConnection` так, чтобы оно позволяло связываться с таблицей employees.gdb, и установим свойство `Connected` объекта `SQLConnection1`, равным `true`, чтобы проверить возможность подключения к базе данных employees.gdb.

Следующим звеном в создании приложения для работы с базами данных InterBase является компонент `SQLDataSet`, реализующий однонаправленный набор данных. Свойству `SQLConnection` этого компонента следует присвоить ссылку на объект `SQLConnection1`, а свойству `CommandText` — текст SQL-запроса, например:

```
select * from employees
```

Если свойству `Connected` объекта `SQLConnection1` присвоено значение `true`, то при назначении свойству `Active` объекта `SQLDataSet1` значения `true`, запрос, введенный в поле `CommandText`, будет отправлен серверу баз данных. Таким способом мы можем проверить корректность SQL-запроса еще на этапе разработки приложения.

Далее следует разместить в форме компонент `DataSetProvider`, являющийся посредником между компонентами DbExpress и клиентскими компонентами доступа к данным (`ClientDataSet` и `DataSource`). Свойство `DataSet` объекта `DataSetProvider1` должно ссылаться на объект `SQLDataSet1`.

Теперь внесем в форму приложения компонент `ClientDataSet`, реализующий клиентский набор данных. Свойству `ProviderName` этого компонента мы присвоим ссылку на компонент `DataSetProvider1`, а свойству `Active` присвоим значение `true`.

В наше приложение необходимо внести еще компонент-источник данных (`DataSource`), который будет посредником между компонентами доступа к данным и визуальными элементами графического интерфейса, отображающими эти данные. Свойство `DataSet` объекта `DataSource1` должно ссылаться на объект, представляющий клиентский набор данных, `ClientDataSet1`.

Таким образом, мы создали всю цепочку компонентов, выполняющих загрузку информации из базы данных. Осталось только добавить визуальный компонент, осуществляющий вывод информации. В нашем приложении эту роль сыграет компонент `DBGrid`, позволяющий отображать содержимое базы данных в форме таблицы. Добавим этот компонент в наше приложение. Свойству `DataSource` объекта `DBGrid1` следует присвоить ссылку на объект `DataSource1`. Поскольку в процессе разработки все компоненты приложения находятся в активном состоянии, мы можем увидеть отображение информации из базы данных еще до запуска приложения. Если визуальный компонент `DBGrid` (рис. 12.2) заполняется данными о сотрудниках, введенными при создании таблицы employees, значит приложение создано успешно.

Рис. 12.2. Форма приложения для работы с базой данных InterBase на этапе разработки

Приложения, взаимодействующие с базами данных InterBase, должны распространяться вместе с библиотекой libsqlib, содержащей соответствующий драйвер.

Глава 13

Работа с СУБД MySQL

Несомненным преимуществом СУБД MySQL (далее MySQL) является ее общедоступность. Если у пользователя есть ОС Linux, то, скорее всего, есть и MySQL. MySQL — это система управления реляционными базами данных с открытым исходным текстом. Как видно из названия, управление базами данных осуществляется в MySQL на основе языка SQL. Практически на всех ОС Linux MySQL позволяет создавать файлы баз данных размером до 2 Гбайт (на некоторых системах максимальный размер файла может составлять 4 Гбайт).

СУБД MySQL включает в себя сервер, осуществляющий управление базами данных, и ряд вспомогательных программ, среди которых присутствуют утилиты настройки и консольная программа SQL-монитор, позволяющая выполнять в консольном режиме основные команды для работы с СУБД. При написании этой главы была использована версия MySQL 3.23. Приведенные здесь и далее советы и инструкции применимы при работе со всеми версиями MySQL, начиная с версии 3.22.11.

Установка и настройка СУБД MySQL

Поскольку СУБД MySQL входит в состав практически каждого Linux-дистрибутива, ее можно установить прямо в процессе установки операционной системы. Если вы не установили MySQL сразу, то можем сделать это позднее, либо из RPM-пакетов, либо из набора исходных текстов, которые, как и документацию по MySQL, можно загрузить с сайта **www.mysql.com**. Там же можно получить информацию и о разработчике MySQL — компании MySQL AB.

Если мы установили MySQL самостоятельно, то, прежде всего, нам необходимо запустить утилиту mysql_install_db, которая выполнит установку стандартных баз данных, необходимых для работы сервера. Запустить сервер MySQL "вручную" в фоновом режиме можно при помощи команды:

```
safe_mysqld &
```

После этого нам следует установить пароль для пользователя MySQL root. Установка пароля для `root` выполняется в режиме `root` командой:

```
mysqladmin -u root password 'root_password'
```

где `root_password` — пароль для пользователя root.

Примечание

Следует особо подчеркнуть, что пароль пользователя root в MySQL в целях безопасности, не должен совпадать с паролем пользователя root в ОС Linux.

Остановить запущенный в фоновом режиме сервер MySQL можно с помощью команды:

```
mysqladmin shutdown -u root -p
```

Ключ `-u` в этой команде, как и в других, позволяет указать имя пользователя MySQL, а ключ `-p` означает, что пользователь готов предоставить пароль, подтверждающий его права. В ответ на запрос пароля введем пароль, установленный нами для пользователя MySQL root. После этого команда `shutdown`, завершающая работу сервера, будет выполнена. Следует еще раз подчеркнуть, что поскольку пароль MySQL root никак не связан с паролем пользователя root ОС Linux, вводить этот пароль придется даже в том случае, если мы работаем в режиме root.

Теперь можно создать базу данных. Запускаем монитор MySQL командой:

```
mysql -u root -p
```

и в ответ на запрос пароля вводим пароль пользователя MySQL root. Если сервер MySQL запущен и работает нормально, то после ввода пароля должно появиться приглашение:

```
mysql>
```

Создадим новую базу данных catalog и в ней таблицу pricelist (листинг 13.1).

Листинг 13.1. Создание базы данных MySQL

```
mysql> create database catalog;
mysql> use catalog;
Database changed
mysql> create table pricelist (Name VARCHAR(32), Section VARCHAR(10),
    -> Manufacturer VARCHAR(15), Price INT(5), Discount INT(2));
Query OK, 0 rows affected (0.02 sec)
```

Мы создали базу данных catalog, содержащую таблицу pricelist для хранения списка товаров торгового предприятия. Значения полей таблицы перечислены в следующем списке:

- `Name` — наименование товара;
- `Section` — категория, к которой относится данный товар;
- `Manufacturer` — сведения о производителе товара;
- `Price` — цена товара;
- `Discount` — скидка.

Пользователь MySQL root имеет полный доступ к этой базе данных (как и ко всем остальным базам MySQL на данном сервере), но регистрироваться каждый раз как root для работы с отдельной базой данных неудобно (и не безопасно). Лучше всего создать новую учетную запись пользователя MySQL и передать этому пользователю права на управление базой данных catalog. Следующая команда MySQL выглядит почти как обычная фраза на английском языке:

```
mysql> grant all privileges on catalog.* to andy@localhost identified by↵
'letmein';
```

Этой командой мы создали учетную запись пользователя andy@localhost и передали новому пользователю все права на работу с базой catalog. Пользователь andy@localhost является пользователем MySQL. Его имя может соответствовать или не соответствовать имени пользователя ОС Linux, для которого создана эта учетная запись (на самом деле любой зарегистрированный пользователь ОС Linux может работать с сервером MySQL под именем andy@localhost, если, конечно, он знает пароль). Пароль, разумеется, может быть совершенно произвольным и в целях безопасности не должен совпадать с паролями пользователей ОС Linux.

Примечание

Более подробные сведения о выборе паролей для пользователей, их безопасном хранении и управлении ими можно найти в оригинальной документации по MySQL. Данная глава должна рассматриваться читателем как ознакомительная и не претендующая на предоставление всех необходимых сведений по вопросам безопасной работы с MySQL.

Теперь вызов монитора MySQL для работы с базой данных catalog может выглядеть следующим образом:

```
[andrei@localhost andrei]$ mysql -u andy -p catalog
```

Здесь слово, следующее после ключа `-p`, является не паролем, а именем базы данных, с которой собирается работать пользователь. Если имя базы данных не было указано в командной строке, его можно указать после входа в программу mysql с помощью команды

```
use database_name
```

где `database_name` — имя базы данных.

После ввода этой команды монитор MySQL выдаст запрос на ввод пароля пользователя andy. Если мы хотим вводить пароль в одной строке с командой вызова монитора, команда должна иметь вид:

```
[andrei@localhost andrei]$ mysql -u andy -pletmein catalog
```

т. е. пароль должен следовать за ключом `-p` без пробела. Такой способ ввода может показаться нам более удобным, но нужно помнить, что при этом пароль вводится, отображаясь на экране терминала.

Заполнить таблицу pricelist можно при помощи команд SQL, работая в мониторе, но можно также воспользоваться данными, хранящимися в текстовом файле. Если у нас есть текстовый файл, в котором данные для таблицы

записаны таким образом, что каждой строке таблицы соответствует одна строка текстового файла (заканчивающаяся символом перевода на новую строку), а элементы данных, соответствующие ячейкам таблицы, разделены символами табуляции, мы можем внести данные из этого файла в таблицу при помощи команды `LOAD DATA INFILE`, например:

```
mysql> LOAD DATA LOCAL INFILE "pricelist.txt" INTO TABLE pricelist;
```

При этом предполагается, что файл, из которого вносятся данные, находится в том же каталоге, что и база данных. Команда `LOAD DATA INFILE` позволяет задавать ряд дополнительных параметров, определяющих формат хранения данных в текстовом файле и его расположение. Более подробные сведения об этой команде можно найти в документации по MySQL.

Для заполнения созданной таблицы удобно также воспользоваться программой DBxExplorer из каталога kylix3/examples/c/db/dbxexplorer/ или kylix3/examples/delphi/db/dbexplorer/. Перед запуском программы щелкнем кнопкой мыши на пиктограмме компонента `SQLConnection` в форме приложения, в открывшемся окне в списке **Connection Name** выберем пункт **MySQLConnection** и заполним таблицу **Connection Settings**, как показано на рис. 13.1. Теперь вы можем работать с базой данных catalog из приложения DBxExplorer. Для того, чтобы приложение DBxExplorer и другие приложения баз данных, создаваемые в Kylix, могли взаимодействовать с сервером MySQL, необходимо выполнить еще одно подготовительное действие. Драйвер MySQL, используемый в Kylix, ссылается на библиотеку libmysqlclient.so, но, скорее всего, даже если сервер MySQL установлен, такого файла в нашем каталоге /usr/lib/ нет. Имя файла библиотеки libmysqlclient содержит номер версии, и чтобы Kylix-приложения могли обращаться к этой библиотеке, нам следует создать символическую ссылку на файл, например:

```
ln -s /usr/lib/libmysqlclient.so.10 /usr/lib/libmysqlclient.so
```

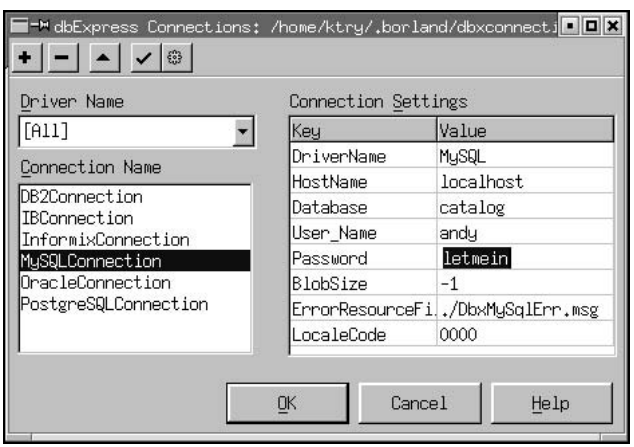


Рис. 13.1. Параметры соединения MySQLConnection

Создание приложения просмотра БД

В этом разделе мы продемонстрируем во всей красе возможности визуального программирования в среде Borland Kylix. Для создания Web-приложения для просмотра базы данных catalog нам потребуются три компонента, пара щелчков мышью и не одной строчки кода на языке C++ или Delphi Language!

Создадим заготовку приложения для Web-сервера (страница **New** диалогового окна **New Items**, пункт **Web Server Application)**. В форме Web-модуля (рис. 13.2) разместим компоненты `SQLConnection` и `SQLDataSet` со страницы **dbExpress** палитры компонентов и `DataSetTableProducer` со страницы **Internet**.

Щелкним кнопкой мыши на пиктограмме `SQLConnection1` в форме модуля. В открывшемся окне в списке **Driver Name** выберем пункт **MySQL**. Отредактируем таблицу **Connection Settings**, как показано на рис. 13.1, и нажмем кнопку **OK**. Мы сразу же можем проверить результат соединения с сервером MySQL, присвоив в инспекторе объектов свойству `Connected` объекта `SQLConnection1` значение `true`. При этом должно открыться диалоговое окно с запросом на ввод пароля для пользователя andy. Если после ввода пароля не возникнет никаких исключительных ситуаций, значит соединение установлено успешно.

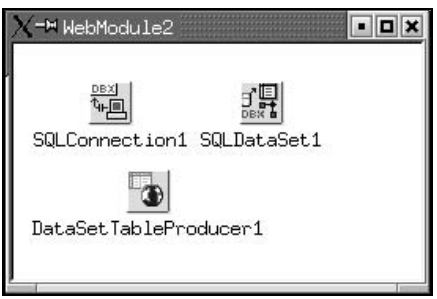


Рис. 13.2. Форма Web-модуля

В инспекторе объектов назначим свойству `SQLConnection` объекта `SQLDataSet1` ссылку на объект `SQLConnection1`. Щелкнем кнопкой мыши в инспекторе объектов на кнопке с многоточием напротив свойства `CommandText`. При этом откроется окно редактора **CommandText Editor** (рис. 13.3), в котором можно редактировать таблицы базы данных MySQL и их поля и задавать текст SQL-запросов. В окне ввода **SQL** введем текст:

```
select * from pricelist
```

и нажмем кнопку **OK**.

После этого в инспекторе объектов можно присвоить значение `true` свойству `Active` объекта `SQLDataSet1`. Теперь назначим свойству `DataSet` объекта `DataSetTableProducer1` ссылку на объект `SQLDataSet1`.

Мы создаем CGI-приложение и нам необходим хотя бы один CGI-сервис (action). Создадим объект `WebActionItem`, как это было описано в главе 7. Свойству `PathInfo` этого объекта присвоим значение "/", свойству `Default` — значение `true`, а свойству `Producer` — ссылку на объект `DataSetTableProducer1`.

На этом работу над приложением можно было бы считать законченной, но с помощью компонента `DataSetTableProducer1` можно внести элементы декоративного оформления в текст генерируемой страницы.

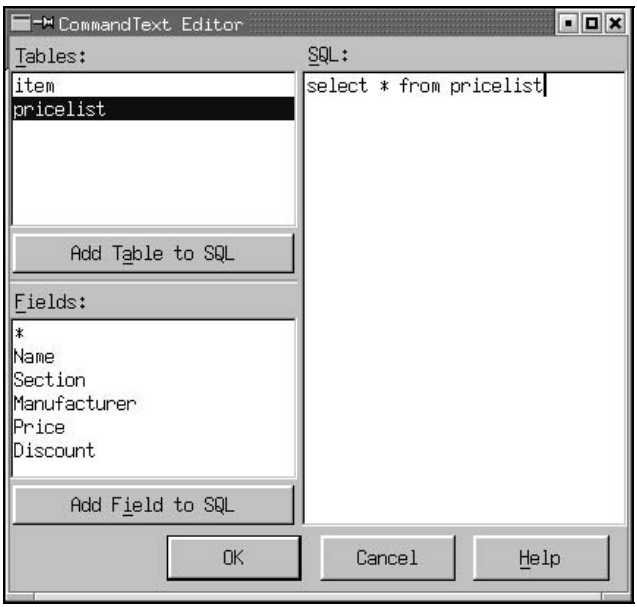


Рис. 13.3. Окно **CommandText Editor**

В инспекторе объектов дважды щелкните кнопкой мыши в поле свойства `Columns` объекта `DataSetTableProducer1`. Это свойство позволяет редактировать внешний вид генерируемой объектом HTML-таблицы. В результате откроется окно специального редактора, позволяющего настроить внешний вид таблицы (рис. 13.4). Отредактируем поля этого окна по своему вкусу. После этого щелкнем мышью в поле свойства `Header` объекта `DataSetTableProducer1`, и в открывшемся окне редактора введем текст:

```
<H2>Catalog Database</H2>
```

В аналогичном окне для свойства `Footer` вводим текст:

```
<HR>
```

Указанные строки, содержащие HTML-код, будут выведены соответственно перед и после HTML-страницы. Заметим, что на этом возможности компонента `DataSetTableProducer` отнюдь не ограничиваются. Компонент позво-

ляет настраивать внешний вид отдельных ячеек таблицы в зависимости от их содержимого.

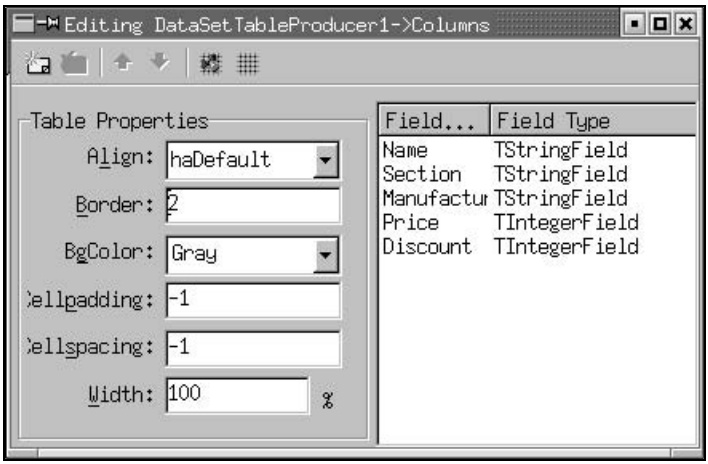
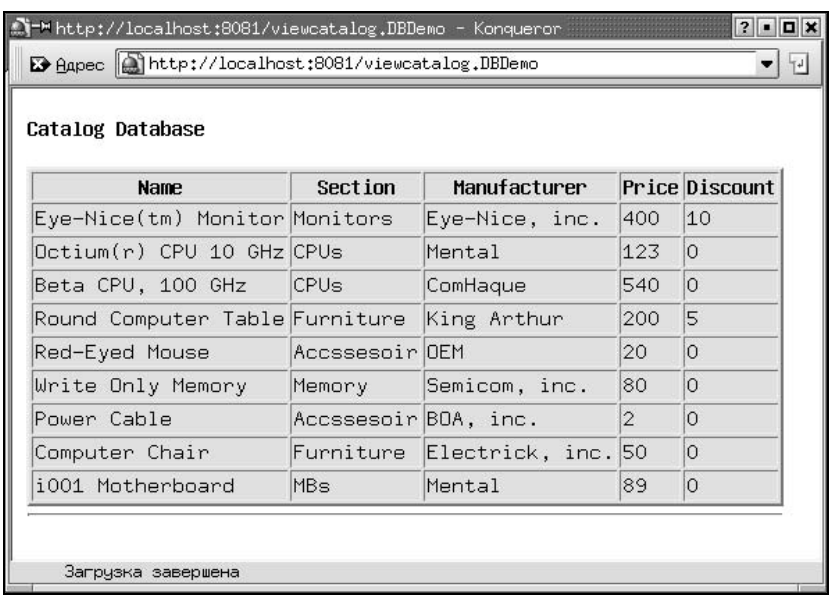


Рис. 13.4. Окно настройки внешнего вида HTML-таблицы

Теперь проект можно скомпилировать. В результате выполнения созданного приложения в окне браузера должна отображаться таблица, подобная той, что показана на рис. 13.5.



Name	Section	Manufacturer	Price	Discount
Eye-Nice(tm) Monitor	Monitors	Eye-Nice, inc.	400	10
Octium(r) CPU 10 GHz	CPUs	Mental	123	0
Beta CPU, 100 GHz	CPUs	ComHaque	540	0
Round Computer Table	Furniture	King Arthur	200	5
Red-Eyed Mouse	Accssesoir	OEM	20	0
Write Only Memory	Memory	Semicom, inc.	80	0
Power Cable	Accssesoir	BOA, inc.	2	0
Computer Chair	Furniture	Electrick, inc.	50	0
i001 Motherboard	MBs	Mental	89	0

Рис. 13.5. Просмотр содержимого базы данных MySQL в окне браузера

Приложения, взаимодействующие с базами данных MySQL, должны распространяться вместе с библиотекой libsqlmy, реализующей необходимый драйвер.

MySQL API

А как же приложения баз данных взаимодействуют с сервером MySQL? Приложения баз данных обращаются к уже упоминавшейся библиотеке mysqlclient. В этой библиотеке реализован набор функций, называемых MySQL API. Компоненты Kylix DataCLX инкапсулируют вызовы MySQL API, однако иногда нам может понадобиться обратиться к этим функциям непосредственно. MySQL API позволяет выполнять некоторые операции, которые недоступны компонентам Kylix, например, завершение работы сервера. Возможны и другие ситуации, когда прямое обращение к функциям MySQL API необходимо. Функции MySQL API декларируются в файле mysql.h, который, как правило, расположен в каталоге /usr/lib/mysql/.

Мы рассмотрим пример программирования MySQL API в Kylix C++ IDE. Для того, чтобы связать приложение с библиотекой libmysqlclient, с помощью команды меню **Project|AddToProject** добавим в проект файл /usr/lib/mysqlclient.so или /usr/lib/mysqlclient.so.10.

Исходный текст фрагмента приложения, использующего MySQL API, приводится в листинге 13.2.

Листинг 13.2. Использование MySQL API

```
#include <mysql/mysql.h>
...
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  st_mysql * MySQL = NULL;
  MySQL = mysql_init(MySQL);
  mysql_real_connect(MySQL, "localhost", "andy", "letmein", "catalog", 0,
  NULL, 0);
  char * ServerStatus = mysql_stat(MySQL);
  Memo1->Text = (String)ServerStatus;
  mysql_close(MySQL);
}
//---------------------------------------------------------------------------
```

Обработчик `Button1Click` заполняет объект `Memo1` текстом, возвращенным функцией `mysql_stat`. Все функции MySQL API используют указатель на структуру `st_mysql`, являющуюся идентификатором (handle) соединения с сервером. Эта структура создается функцией `mysql_init`, а уничтожается функцией `mysql_close`.

Следует отметить одну важную особенность MySQL API. Все блоки памяти, выделяемые функциями MySQL API, даже строки, возвращаемые функциями, принадлежат библиотеке libmysqlclient, и не следует пытаться освободить эту память самостоятельно.

Глава 14

Язык запросов SQL и компоненты dbExpress

Из самих названий компонентов панели **dbExpress** следует, что при взаимодействии с базами данных (БД) они используют язык запросов SQL. Хотя модель программирования приложений в Borland Kylix скрывает многие детали взаимодействия между приложениями и базами данных, при разработке приложений БД вам все же придется иметь дело с SQL-запросами. Описания двух реализаций СУБД, основанных на языке SQL, были приведены в главах 12 и 13. В данной главе, посвященной использованию языка SQL при работе с компонентами dbExpress, осуществляющими связь с самыми разными базами данных, мы приведем краткое "программно-независимое" описание этого языка.

Примечание

Эта книга не претендует на роль руководства по языку SQL. В следующем кратком разделе приводятся лишь те сведения о языке SQL, которые необходимы для построения приложений баз данных, рассматриваемых в этой книге. Впрочем, этих сведений может оказаться вполне достаточно для профессиональной работы с несложными базами данных. Более подробные сведения о языке SQL, включая формальные описания конструкций языка, можно найти, например, в документации, поставляемой вместе с СУБД Borland InterBase. Там же описываются ограничения и расширения языка SQL, присущие этой СУБД. Вообще, выбирая какую-либо СУБД в качестве основы базы данных, следует выяснить особенности реализации языка SQL в этой СУБД.

Введение в язык запросов SQL

Язык SQL (Structured Query Language, язык структурированных запросов) позволяет формировать запросы на получение информации из реляционных баз данных. Помимо операторов формирования запросов и манипулирования базами данных язык SQL включает и другие возможности, среди которых стоит отметить средства определения схемы БД и манипулирования этой схемой и средства управления транзакциями. Иначе говоря, язык SQL является средством, позволяющим полностью описать реляционную модель данных.

Типы данных в языке SQL

Как уже отмечалось, в рамках реляционной модели БД все типы данных должны быть простыми, т. е. на уровне этой модели тип данных нельзя логически разделить на составляющие элементы (хотя вне модели этот тип может рассматриваться как сложный). В соответствии с этим в языке SQL значение любого типа не может быть логически разбито на другие значения.

Значения типов данных могут быть определенными или неопределенными. *Неопределенное значение* — это зависящее от реализации значение, которое гарантированно отлично от любого определенного значения соответствующего типа. Можно считать, что имеется всего одно неопределенное значение, входящее в любой тип данных языка SQL. Для неопределенного значения отсутствует представляющий его литерал, хотя в некоторых случаях для выражения этого значения используется ключевое слово `NULL`. Пустые значения играют важную роль в языке SQL, так как в реальной жизни при заполнении таблиц баз данных не всегда можно получить все значения, требуемые в рамках реляционной модели данных.

Основными типами данных в языке SQL являются следующие: `CHARACTER`, `CHARACTERVARYING`, `BIT`, `BITVARYING`, `NUMERIC`, `DECIMAL`, `INTEGER`, `SMALLINT`, `FLOAT`, `REAL`, `DOUBLEPRECISION`, `DATE`, `TIME`, `TIMESTAMP` и `INTERVAL`.

При указании типов в конструкциях языка SQL допускаются сокращения названий типов, например, тип `CHARACTER` можно указывать как `CHAR`, тип `CHARACTERVARYING` — как `VARCHAR`, тип `INTEGER` — как `INT`. Стоит отметить, что токены (синтаксические элементы) языка SQL являются регистронезависимыми (как в языке Delphi Language).

Типы данных `CHARACTER` и `CHARACTERVARYING` называются типами данных *символьных* строк; типы данных `BIT` и `BITVARYING` — типами данных *битовых* строк. Типы данных символьных и битовых строк носят общее название *строчных типов*.

Разница между типами `CHARACTER` и `CHARACTERVARYING` заключается в том, что тип `CHARACTER` позволяет задать строку фиксированной длины, тогда как тип `CHARACTERVARYING` позволяет определить строку переменной длины с ограниченным максимальным числом символов. Например, декларация:

```
CHARACTER(10)
```

определяет строку из 10 символов, а декларация:

```
CHARACTERVARYING(255)
```

определяет строку переменной длины, которая может включать не более чем 255 символов. Таковы же различия между типами `BIT` и `BITVARYING`, только в этом случае указывается число не символов, а битов.

Следует запомнить общее правило при работе с типами данных языка SQL: значение типа может иметь переменный размер, но максимальный размер

должен быть определен заранее. Типы неопределенного размера, такие как строки с нулевым терминантным символом, в языке SQL не допускаются.

Во многих реализациях SQL численные типы (`NUMERIC`, `DECIMAL`, `INTEGER`, `SMALLINT`, `FLOAT`, `REAL`) позволяют дополнительно указывать точность. Например, декларация:

```
INT(5)
```

указывает пятизначное целое число.

Операции над различными типами данных

В языке SQL определен ряд операций, которые можно выполнять над данными различных типов.

К операциям, выполняемым над символьными строками, относятся следующие:

- оператор конкатенации возвращает символьную строку, произведенную путем соединения строк-операндов в том порядке, как они заданы;
- функция выделения подстроки `SUBSTRING` принимает три параметра — строку, номер начальной позиции и длину, и возвращает строку, выделенную из строки-параметра в соответствии со значениями двух последних параметров;
- функция `UPPER` возвращает строку, в которой все буквы строки-параметра из нижнего регистра заменяются на буквы верхнего регистра;
- функция `LOWER` заменяет в заданной строке все символы верхнего регистра на символы нижнего регистра;
- операция определения длины строки возвращает длину заданной символьной строки в символах, октетах или битах (в зависимости от вида вычисляющей функции) в виде целого числа;
- операция определения позиции вхождения определяет первую позицию вхождения строки S1 в строке S, если строка S1 не входит в строку S, то возвращается нуль.

Над битовыми строками допустимы следующие операции:

- операция битовой конкатенации возвращает результирующую битовую строку, полученную путем конкатенации строк-операндов в том порядке, в котором они заданы;
- функция извлечения подстроки из битовой строки. Синтаксис этой функции идентичен синтаксису функции `SUBSTRING` для символьных строк за исключением того, что первый аргумент и возвращаемое значение являются битовыми строками;
- операция определения длины возвращает длину заданной битовой строки в октетах или битах в зависимости от выбранной функции;

- операция определения позиции вхождения определяет первую позицию вхождения битовой строки S1 в строке S, если строка S1 не входит в строку S, то возвращается нуль.

Над типами данных даты и времени (`DATE`, `TIME`, `TIMESTAMP`, `INTERVAL`) можно выполнять следующие операции:

- операция сложения двух дат или даты и интервала возвращает значение даты;
- операция вычитания двух дат возвращает значение типа `INTERVAL`, а операция вычитания интервала из даты возвращает значение даты.

Над численными типами данных допустимы следующие операции:

- арифметические операции (сложение, вычитание, умножение, деление);
- различные математические функции (тригонометрические, логарифмические, статистические и т. п.).

Общие замечания о командах языка SQL

Как уже отмечалось большинство команд языка SQL должно заканчиваться символом ";". Некоторые команды не нуждаются в завершающем символе ";", но допускают его. В большинстве программ, позволяющих вводить команды языка SQL, реализована возможность многострочного ввода. Это означает, что, если мы нажмем клавишу <Enter> в конце команды, не введя символ завершения, программа предложит нам продолжить ввод команды. При работе в локальной СУБД нам обычно не нужно вводить специальной команды для завершения транзакции, в распределенных же системах, с которыми одновременно могут работать несколько пользователей, завершать транзакцию следует специальной командой `COMMIT`.

Домены

Определение понятия домена было дано в предыдущей главе, там же было сказано о той роли, которую домены играют в реляционной модели баз данных. К этому следует добавить еще одно практическое удобство использования доменов. Если в базе данных существует несколько таблиц, содержащих одинаковые структурные и смысловые определения полей данных, удобно, вместо того чтобы каждый раз вводить громоздкую декларацию поля данных, определить соответствующий домен и использовать имя домена при создании таблицы.

Домены создаются при помощи оператора `CREATE DOMAIN`. Формальное описание синтаксиса оператора определения (создания) домена приводится в листинге 14.1.

Листинг 14.1. Синтаксис оператора создания домена

```
<domain definition> ::=
       CREATE DOMAIN <domain name> [ AS ] <data type>
       [ <default clause> ]
       [ <domain constraint>... ]
       [ <collate clause> ]
<domain constraint> ::=
       [ <constraint name definition> ]
       <check constraint definition> [ <constraint attributes> ]
```

Приведем несколько примеров создания доменов:

```
create domain first_name_t as varchar(10) not null;
```

В этой строке мы создаем домен с именем `first_name_t`, имеющий тип данных "строка переменной длины с максимальным числом символов — 10", и указываем, что элементы данного домена не могут быть пустыми (напомним, что домен — это подмножество множества элементов данных некоторого типа). Теперь в описании полей таблицы можно использовать идентификатор `first_name_t`.

Еще один пример определения домена:

```
create domain employee_salary_t int default 0;
```

Этой командой мы создаем домен `employee_salary_t` (зарплата сотрудника) типа `integer` и указываем, что значением по умолчанию для данного домена должно быть число 0. В этом определении нам не нужно указание `not null`, так как, если значение поля, принадлежащего данному домену, не будет указано явным образом, этому полю будет присвоено значение 0.

Создадим еще один домен:

```
create domain employee_age_t as int not null check (value > 13 and
  value < 100);
```

Столбцы таблицы, принадлежащие этому домену, должны содержать возраст сотрудника. В определении домена мы указываем, что значение соответствующего элемента не может быть пустым и, кроме того, задаем ограничения возможных значений. В нашем случае значением возраста должно быть число больше 13 и меньше 100. Введение подобных ограничений при работе с доменами очень полезно, так как позволяет лучше определить смысл данных, для которых создается домен и избежать многих ошибок при вводе/обработке данных. Естественно, вводимые ограничения должны соответствовать смыслу данных в рамках конкретной модели. Мы можем расширить диапазон значений возрастов, если предположим, что в нашей компании могут работать сотрудники старше 100 лет, а вот указание значения по умолчанию для этого поля вряд ли имело бы смысл.

Создание таблиц

В каждой базе данных, основанной на языке SQL, может быть определена одна или несколько таблиц, в которых и хранятся данные. Таблицы баз данных создаются при помощи команды `CREATE TABLE`. Синтаксис команды `CREATE TABLE` выглядит следующим образом:

```
CREATE TABLE table [EXTERNAL [FILE] " <filespec>"]
(<col_def> [, <col_def> | <tconstraint> ...]);
```

Здесь `table` — имя таблицы, уникальное в пределах базы данных. Необязательный элемент `EXTERNAL` позволяет заполнять таблицу данными из текстового файла *(подробную информацию см. в гл. 12)*. Далее следуют определения столбцов таблицы, очень похожие на описания доменов (домены можно использовать в описаниях столбцов).

Приведем примеры создания нескольких таблиц:

```
create table simple(column1 int not null primary key, column2 char(10));
```

В этом примере мы создаем таблицу с именем `simple` и двумя столбцами — `column1` типа `integer` и `column2` типа "строка из 10 символов". Первый столбец таблицы помечен как *первичный ключ*. Первичные ключи являются уникальными значениями, идентифицирующими записи таблицы. SQL-СУБД используют первичные ключи для автоматической индексации таблиц, так, что поиск записи по первичному ключу занимает сравнительно немного времени. По умолчанию, если не указан другой критерий сортировки, записи в таблицах сортируются именно по значению первичных ключей. Использование первичных ключей в таблицах не обязательно, но желательно. Столбцы таблицы, хранящие первичные ключи, должны удовлетворять определенным условиям. Во-первых, эти столбцы не должны содержать пустых значений (обратите внимание на указание `not null` в определении столбца `column1`). Во-вторых, все значения в таких столбцах должны быть уникальными.

Существует и другой способ указания первичных ключей. Та же самая таблица могла бы быть создана командой:

```
create table simple(column1 int not null, column2 char(10), ↳
primary key(column1));
```

Из ограничений, налагаемых на первичные ключи, следует, что лучше всего для этого подходят столбцы таблицы, содержащие числовые или буквенно-цифровые идентификаторы записей, хотя в качестве первичных ключей можно использовать любые столбцы с гарантированной уникальностью значений.

Рассмотрим еще один пример создания таблицы:

```
create table employees(surname surname_t, name first_name_t, ↳
birthdate date, salary salary_t);
```

Эта команда создает таблицу employees с четырьмя столбцами — name, surname, birthdate и salary. Столбцы name, surname и salary используют определения доменов. В объявлении типов данных столбцов таблицы можно использовать те же элементы, что и в объявлении домена. Мы можем использовать указания not null, default и ограничения допустимых значений. Важным элементом описания столбцов таблицы является указание unique. Если в описании столбца таблицы используется указание unique, это означает, что в столбце не может быть двух полей с одинаковыми значениями.

Для упрощения поиска в таблицах баз данных СУБД используют индексацию записей. Мы можем указать, какие столбцы таблицы (их может быть несколько) используются для индексации с помощью команды create index, например, команда:

```
create index first_index on employees(surname);
```

создает новый индекс с именем first_index для таблицы employees. Индексация должна производиться по значениям столбца surname.

Для заполнения таблиц данными используется SQL-команда insert. Эта команда позволяет добавить в таблицу одну запись. Формат команды следующий:

```
INSERT INTO <table_name> [(col [, col ...])]
{VALUES (<val> [, <val> ...]) | <select_expr>};
```

Здесь table_name — имя таблицы, уникальное в пределах базы данных, элементы col — указания столбцов таблицы, в которые вносятся данные, элементы <val> — вносимые значения. Вместо перечня значений в команде INSERT можно использовать выражение SELECT (<select_expr>), позволяющее перенести в данную таблицу значения из другой таблицы. Более подробное описание операции SELECT будет дано в следующем разделе этой главы. В качестве примера использования операции INSERT приведем команду добавления записи в таблицу employees.

```
insert into employees (surname, name, birthdate, salary)↳
  values('Иванов', 'Иван' '7-FEB-1970', 100);
```

Если порядок следования значений, вносимых в таблицу, точно соответствует порядку столбцов, как это и было в приведенном выше примере, перечень столбцов таблицы можно опустить:

```
insert into employees values('Иванов', 'Иван', '7-FEB-1970', 100);
```

Для удаления записей из таблицы служит команда DELETE. Этой команде необходимо указать имя таблицы, из которой выполняется удаление, а также критерии выбора удаляемых записей.

Примечание

Выполнение команды DELETE без указания критериев выбора записей приведет к удалению всех записей из таблицы!

Например, следующая команда удаляет из таблицы employees все записи, у которых поле salary содержит значение 0.

```
delete from employees where salary = 0;
```

Модификация уже существующих в таблице записей выполняется при помощи команды UPDATE.

Например, команда:

```
update employees set salary = 200 where surname = 'Иванов' and ↵
  birthdate = '7-FEB-1970';
```

изменяет значение поля salary для записи сотрудника по фамилии Иванов с датой рождения 7-го февраля 1970 года. Следует учесть, что если в таблице окажется несколько записей, соответствующих указанным критериям, значения поля salary будет изменено во всех этих записях.

Выборка записей из таблиц

Выборка данных из таблиц выполняется командой SELECT. Команде SELECT передается имя таблицы, из которой следует выбрать данные, а также ряд параметров, указывающих, какие именно данные необходимо выбрать и как они должны быть представлены. Результатом выполнения команды SELECT является либо таблица, либо пустое множество, которое возвращается, если исходная таблица пуста, либо если указанным в команде критериям выбора не соответствует ни одна запись таблицы.

Команда SELECT обладает чрезвычайно широким набором дополнительных параметров и формальное описание ее синтаксиса выглядит довольно громоздко, так что приводить здесь мы его не будем. Вместо этого рассмотрим несколько примеров использования команды SELECT.

Простейший вариант использования команды SELECT:

```
select * from employees;
```

возвращает таблицу, содержащую все записи таблицы employees. Если мы внимательно посмотрим на данное выше определение таблицы employees, то увидим, что эта таблица может содержать полностью совпадающие записи. Для того чтобы в результирующую таблицу повторяющиеся записи включались лишь один раз, следует использовать команду:

```
select * distinct from employees;
```

Команда:

```
select * from employees where salary > 100 and salary < 500;
```

вернет таблицу, содержащую записи таблицы employees, в которых значение salary лежит в пределах между 100 и 500.

Если нам нужны только фамилии и значения зарплаты сотрудников, чья зарплата превышает 100, воспользуемся командой:

```
select surname, salary from employees where salary > 100;
```

Возвращенная в результате выполнения этой команды таблица будет содержать только два столбца — `surname` и `salary`. Таким образом, символ групповой операции "*" после имени команды `SELECT` обозначает "все столбцы данной таблицы".

Если мы хотим, чтобы в результирующей таблице записи были отсортированы по значению поля `salary`, воспользуемся командой:

```
select surname, salary from employees where salary > 100 order by salary;
```

Если в базе данных, с которой мы работаем, определено несколько таблиц, мы можете перечислить несколько имен таблиц после ключевого слова `from`.

Некоторые другие команды языка SQL

Выше мы упоминали о том, что в базах данных часто используются поля, хранящие значения уникальных численных идентификаторов записей. Задавать такие значения "вручную", как правило, не очень удобно. Для автоматической генерации идентификаторов в языке SQL можно использовать специальные *генераторы*. Генераторы создаются командой:

```
create generator generator_name;
```

где `generator_name` — имя генератора.

Генератор генерирует новое значение, увеличивая свое текущее значение на некоторую величину. После создания нового генератора его текущим значением по умолчанию является значение 0. Если мы хотим изменить это значение, используем команду:

```
set generator generator_name to <value>;
```

где `<value>` — новое зачение.

Рассмотрим пример использования генераторов. Предположим, мы работаем с определенной выше таблицей `simple`, содержащей столбец `id`. Создадим генератор id_generator:

```
create generator id_generator;
```

Для генерации нового значения с помощью созданного генератора используется SQL-функция `gen_id`. Аргументами этой функции являются имя генератора и величина, на которую нужно увеличить его текущее значение. Следующая команда иллюстрирует добавление записи в таблицу `simple` с использованием генератора.

```
insert into simple(column1, column2) values(gen_id(id_generator, 1),↳
'some value');
```

Для уничтожения объектов базы данных используется семейство команд DROP. Ниже перечислены некоторые команды этого семейства:

- drop database <имя базы данных>; — уничтожает базу данных;
- drop domain <имя домена>; — уничтожает домен;
- drop index <имя индекса>; — уничтожает индекс;
- drop table <имя таблицы>; — уничтожает таблицу.

Использование команд *COMMIT* и *ROLLBACK*

Команда COMMIT, как уже отмечалось, завершает текущую транзакцию и "фиксирует" внесенные изменения. Хотя зачастую даже при работе в "низкоуровневом" SQL-клиенте можно обойтись и без этой команды (поскольку большинство программ-клиентов SQL автоматически завершает незавершенные транзакции при выходе из программы), в некоторых случаях она все же необходима. В нашей собственной сессии работы с базой данных все изменения вступают в силу сразу после ввода команд, но другие пользователи, работающие с той же базой данных, могут не видеть их до тех пор, пока мы не введем команду COMMIT. В этом есть смысл, так как пока мы не ввели команду COMMIT, мы можем легко отказаться от всех внесенных в базу данных изменений с помощью команды ROLLBACK, а, значит, эти изменения носят еще "неокончательный" характер. Внесение большого числа изменений, невидимых другим пользователям, чревато возникновением конфликтов, когда разные пользователи пытаются изменить один и тот же объект.

Другая ситуация иллюстрируется следующим примером. Допустим, мы работаем в СУБД InterBase и в SQL-клиенте этой СУБД даем команды:

```
create table mytable(...);
insert into mytable ...;
```

Если теперь мы вдруг захотим отказаться от таблицы mytable при помощи команды:

```
drop table mytable;
```

то в ответ получим сообщение вроде:

```
object mytable is already in use
```

Кто же использует mytable? Мы сами! Для того, чтобы удалить таблицу, следует либо вызвать команду ROLLBACK, либо завершить текущую транзакцию командой COMMIT и затем дать команду на удаление таблицы. Команда ROLLBACK позволяет отменить все операции с базой данных, выполненные с момента последнего ввода команды COMMIT.

Компоненты Kylix, предназначенные для работы с базами данных, как правило, не требуют явного ввода команды COMMIT. Обычно методы, вносящие

изменения в базу данных (такие, как `ApplyUpdates`), автоматически завершают транзакцию и делают внесенные изменения действенными для остальных пользователей.

Видимость результатов транзакций для других пользователей зависит также от режима управления транзакциями, в котором они работают. Пользователь, начавший транзакцию, может не видеть изменения, сделанные другими пользователями после начала своей транзакции, до тех пор, пока не завершит начатую транзакцию.

В других режимах работы данная транзакция может "видеть" результаты выполнения других транзакций, начатых во время ее выполнения, даже если эти другие транзакции еще не завершены.

Команда *SHOW*

Часто бывает необходимо просмотреть список имен объектов определенного типа, существующих в текущей базе данных. Для этой цели служит команда `SHOW`. Ниже приводится ряд примеров использования команды `SHOW`. Эта команда не относится к числу команд языка SQL, но и СУБД InterBase, и СУБД MySQL поддерживают ее.

- Команда `show tables` возвращает список таблиц базы данных.
- Команда `show domains` возвращает список доменов, определенных в базе данных.
- Команда `show generators` возвращает список генераторов, определенных в базе данных.

Компоненты dbExpress

На странице **dbExpress** палитры компонентов Kylix расположены семь компонентов, предназначенных для непосредственного взаимодействия с СУБД, использующими язык SQL для генерации запросов. В этом разделе мы рассмотрим все компоненты dbExpress, обсудим особенности транзакций при использовании dbExpress и напишем приложение, генерирующее динамические SQL-запросы к базе данных. Следует отметить, что на практике нам, скорее всего, не понадобятся все компоненты dbExpress. Наиболее важными из этих компонентов в настоящее время являются компоненты `SQLConnection` и `SQLDataSet`. Такие компоненты, как `SQLQuery` и `SQLStoredProc`, существуют скорее ради обеспечения обратной совместимости с предыдущими версиями Kylix, Delphi и C++ Builder.

Компонент *SQLConnection*

Компонент `SQLConnection` предназначен для установления непосредственной связи с сервером SQL-СУБД. Остальные компоненты dbExpress используют этот компонент для обмена данными с СУБД.

Важнейшей особенностью компонента `SQLConnection` с точки зрения программиста является возможность настроить соединение с СУБД. Отдельные свойства компонента `SQLConnection` позволяют настроить многие параметры соединения, однако проще настроить все сразу с помощью редактора соединений (Connection Editor). Для того, чтобы вызвать редактор соединений, нужно дважды щелкнуть кнопкой мыши на пиктограмме компонента `SQLConnection` в форме приложения. При этом открывается окно редактора, показанное на рис. 14.1.

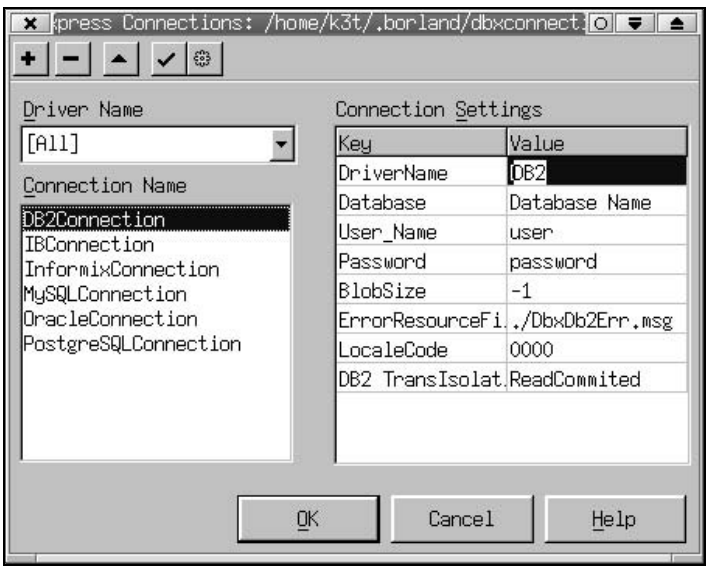


Рис. 14.1. Окно редактора соединений компонента `SQLConnection`

С двумя примерами настройки соединений для работы с конкретными СУБД мы познакомились в главах 12 и 13. Здесь же мы опишем свойства редактора, общие для всех СУБД. Список **Driver Name** позволяет выбрать драйвер для взаимодействия с сервером базы данных. В этом списке мы можем выбрать конкретный драйвер для той базы данных, которую собираемся использовать, или оставить выбранное по умолчанию значение **[All]**.

Список **Connection Name** позволяет выбрать предустановленную конфигурацию соединения. Если в списке **Driver Name** выбран конкретный драйвер, в списке **Connection Name** будут присутствовать только конфигурации соеди-

нений для выбранного драйвера, если же в списке **Driver Name** выбрано значение **[All]**, в списке **Connection Name** будут показаны все предустановленные конфигурации.

Таблица **Connection Settings** содержит список параметров конфигурации соединения. В столбце **Key** содержатся наименования параметров соединения, а в столбце **Value** — их значения. Список параметров соединения зависит от типа базы данных, для связи с которой используется соединение. Ниже мы перечислим параметры, общие для всех соединений.

- `DriverName` — имя драйвера для данного соединения. Соответствует свойству `DriverName` компонента `SQLConnection`.
- `Database` — имя базы данных, формат которого зависит от используемой СУБД. Для СУБД InterBase - это имя gdb-файла, а для СУБД MySQL — имя базы данных, указанное в команде `CREATE DATABASE`.
- `BlobSize` — указывает размер по умолчанию "большого двоичного объекта" (Binary Large OBject, BLOB). Значение 1 указывает, что размер неограничен.
- `User_Name` — имя пользователя базы данных.
- `Password` — пароль пользователя базы данных.
- `LocaleCode` — идентификатор кодовой страницы (локали), соответствует свойству `LocaleCode` компонента `SQLConnection`.

Мы можем создавать собственные предустановленные конфигурации соединения в дополнение к имеющимся. Нажмем кнопку с символом "+" на панели инструментов редактора. В открывшемся диалоговом окне нам будет предложено выбрать драйвер, для которого создается новая конфигурация, и ее имя. После того, как мы нажмем кнопку **OK**, в списке **Connection Name** появится новый пункт, соответствующий созданной конфигурации. Параметры нового соединения, установленные по умолчанию, будут соответствовать параметрам базового соединения для данного драйвера. Создание собственных конфигураций может быть удобно, если мы разрабатываем несколько приложений, работающих с разными базами данных одной и той же СУБД.

Значения параметров соединений хранятся независимо от проектов приложений и доступны всем проектам. Для хранения значений параметров Kylix использует файл dbxconnections, расположенный либо в скрытом подкаталоге .borland, который находится в домашнем каталоге пользователя, установившего Kylix, либо в каталоге /usr/local/etc/dbxconnections.conf, если Kylix устанавливался в режиме root.

Компонент `SQLConnection` содержит методы, позволяющие завершать или отменять выполняемые транзакции. При работе с компонентами dbExpress, нам вряд ли придется иметь дело непосредственно с управлением транзак-

циями, а если такая необходимость и возникнет, можно воспользоваться командами языка SQL. Таким образом, в случае со специальными методами управления транзакциями компонента SQLConnection, мы впервые сталкиваемся с "параллелизмом" функций, когда для операций, которые можно выполнить с помощью команд языка SQL, вводятся специальные методы компонентов. Тем не менее, рассмотрим эти методы.

Для запуска транзакции служит метод StartTransaction, единственным параметром которого является структура TTransactionDesc, описывающая транзакцию. Наиболее важными полями этой структуры являются поле TransactionID, содержащее идентификатор транзакции (который назначается пользователем, причем идентификатор должен быть уникальным в рамках сессии) и поле IsolationLevel, определяющее уровень изоляции транзакции. Возможными значениями этого поля являются:

- xilDIRTYREAD ("грязное" чтение) — для данной транзакции будут видимы все изменения, внесенные другими транзакциями, даже если они еще не подтверждены;
- xilREADCOMMITES — для данной транзакции будут видимы все изменения, подтвержденные до начала данной транзакции;
- xilREPEATABLEREAD — для данной транзакции будут видимы все подтвержденные изменения, в том числе и те, которые были подтверждены после начала данной транзакции.

Методы компонента SQLConnection Commit и Rollback аналогичны по своему действию одноименным командам SQL. Каждому из этих методов в качестве параметра передается структура TTransactionDesc.

Рассмотрим некоторые свойства компонента SQLConnection. При просмотре окна инспектора объектов мы можем заметить, что после настройки соединения с помощью редактора соединений многим свойствам компонента SQLConnection уже присвоены необходимые значения.

Свойство Connected позволяет управлять подключением к базе данных. Компонент SQLConnection подключается к БД при установке значения этого свойства, равным true, и отключается при установке значения false. При активизации набора данных, использующих компонент SQLConnection, этому свойству автоматически присваивается значение true. Если мы присвоим свойству Connected значение true во время разработки программы, компонент SQLConnection немедленно попытается установить связь с базой данных (при этом может появиться диалоговое окно, предлагающее ввести пароль пользователя БД). Таким способом можно проверить правильность настроек соединения.

Особый интерес представляет свойство LoginPrompt. Это свойство управляет выводом диалогового окна, содержащего запрос имени пользователя и пароля для доступа к БД. Если свойству LoginPrompt

и пароля для доступа к БД. Если свойству `LoginPrompt` присвоено значение `true`, соответствующее диалоговое окно появляется в момент запуска программы на исполнение (и даже может появиться в процессе разработки программы, как отмечалось выше). Вообще, если имя пользователя и пароль указаны в настройках соединения, программа уже "знает" эти реквизиты в момент запуска (и во время разработки) и, следовательно, не нуждается в предоставлении их пользователем. Если мы хотим, чтобы программа не запрашивала у пользователя имя и пароль, мы можем установить значение свойства `LoginPrompt`, равным `false`. В этом случае пользователь нашей программы сможет получить доступ к базе данных, не предоставляя необходимых реквизитов. Иногда такой подход может быть оправдан, например, если наша программа предоставляет доступ к БД "только для чтения".

Механизм указания реквизитов пользователя во время настройки соединения с базой данных в редакторе соединений, на первый взгляд, работает достаточно гладко, но с ним связаны две проблемы. Во-первых, в этом случае и имя пользователя, и пароль хранятся в "теле" программы в незашифрованном виде (в этом можно убедиться с помощью любого шестнадцатиричного редактора), а, во-вторых, приложение теряет гибкость. Ведь и пароль, и имя пользователя могут меняться администратором базы данных.

Можно ли сделать так, чтобы программа не хранила реквизиты, необходимые для подключения к базе данных, а получала их от пользователя в момент запуска? В силу ряда причин стандартный метод аутентификации пользователей в этом случае нам не поможет. Можно предложить следующее решение (на примере языка C++).

В редакторе соединений сделаем пустыми значения полей `User_Name` и `Password`. При этом нам придется установить значение свойства `Connected` компонента `SQLConnection`, равным `false`. Это же значение придется присвоить всем свойствам `Active` компонентов-наборов данных (DataSets).

Мы будем вызывать диалог аутентификации пользователя явным образом (а заодно и русифицируем его), так что установим значение свойства `LoginPrompt` компонента `SQLConnection`, равным `false`. Далее добавим в заголовочный файл, в котором объявляется класс главной формы приложения, строку:

```
#include <QDBLogDlg.hpp>
```

В варианте для Delphi Language нам нужно было бы включить модуль `QDBLogDlg` в раздел `uses`. Теперь мы можем работать с диалогом аутентификации пользователя непосредственно. Создадим обработчик события `OnCreate` главной формы и внесем в него текст из листинга 14.2.

Листинг 14.2 Обработчик события `OnCreate`

```
__fastcall TForm1::FormCreate(TObject *Sender)
{
  TLoginDialog * LogDlg = new TLoginDialog(this);
  LogDlg->Caption = "Зарегистрируйтесь";
  LogDlg->DatabaseName->Caption = "Catalog";
  LogDlg->Label1->Caption = "Имя";
  LogDlg->Label2->Caption = "Пароль";
  LogDlg->Label3->Caption = "База данных";
  LogDlg->CancelButton->Caption = "Отмена";
  LogDlg->ActiveControl = LogDlg->UserName;
  LogDlg->ShowModal();
  SQLConnection1->Params->Values["User_Name"] =
  (String)LogDlg->UserName->Text;
  SQLConnection1->Params->Values["Password"] =
  (String)LogDlg->Password->Text;
  delete LogDlg;
  SQLConnection1->Connected = true;
}
```

В этом обработчике мы создаем объект класса `TLoginDialog` и отображаем его как модальное окно. Затем мы заполняем свойство `Params` объекта `SQLConnection1` значениями имени и пароля, введенными пользователем. Свойство `Params` имеет тип `TStringList` и хранит настройки соединений. Настройки хранятся в виде списка строк формата:

```
Имя_Параметра=<значение>
```

где имена параметров и значения соответствуют таблице **Connection Settings** редактора соединений.

Затем мы удаляем объект класса `TLoginDialog` и присваиваем свойству `Connected` компонента `SQLConnection1` значение `true`, что заставляет компонент установить соединение с базой данных.

Для успешной работы с базой данных нам теперь нужно присвоить значение `true` свойствам `Active` компонентов-наборов данных. Удобнее всего сделать это в обработчике события `OnAfterConnect` объекта `SQLConnection1` (листинг 14.3).

Листинг 14.3 Обработчик события `OnAfterConnect`

```
__fastcall TForm1::SQLConnection1AfterConnect(TObject *Sender)
{
  SQLDataSet1->Active = true;
  ClientDataSet1->Active = true;
}
```

Теперь при запуске программы на экране появится диалоговое окно, представленное на рис. 14.2.

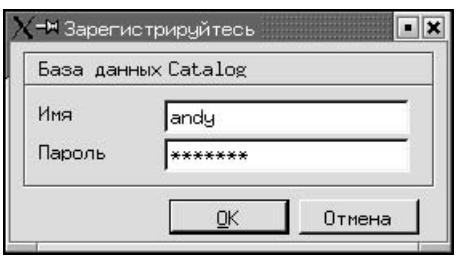


Рис. 14.2. Диалоговое окно аутентификации пользователя базы данных

Если в этом окне пользователь ввел верные реквизиты, он получит доступ к соответствующей базе данных.

С помощью списка `Params` можно настраивать свойства соединения во время выполнения программы точно так же, как это делается в редакторе соединений во время редактирования. Например, строка:

```
SQLConnection1->Params->Values["Database"] = "newbase.gdb";
```

присваивает новое значение параметру `Database`.

Булево свойство `KeepConnection` управляет состоянием соединения с базой данных при отсутствии открытых наборов данных. Если этому свойству присвоить значение `true` (что и сделано по умолчанию), то соединение с базой данных будет сохраняться, даже когда все наборы данных закрыты. Такое поведение компонента уменьшает трафик и ускоряет работу распределенных приложений. Нам также требуется оставить это свойство, равным `true`, если наша программа часто открывает и закрывает наборы данных для одной и той же базы данных.

С помощью свойства `TableScope` можно определить типы таблиц, доступные пользователю. По умолчанию считается, что пользователь может работать с обычными (несистемными) таблицами и запросами (поля `tsTable` и `tsView`).

Получить список компонентов-наборов данных, открытых в данный момент, можно с помощью свойства-массива `DataSets`, доступного во время выполнения программы. Число элементов этого массива можно узнать при помощи свойства `DataSetCount`.

Закрыть все открытые наборы данных можно с помощью метода `CloseDataSets`.

Компонент `SQLConnection` также позволяет передавать команды серверу СУБД. Для передачи серверу SQL-команды служит метод `Execute`. В первом параметре этого метода передается SQL-команда. Второй параметр — указатель на объект `TParams` позволяет передавать параметры SQL-команды. Использование отдельного компонента для передачи параметров SQL-команды не обязательно, но очень удобно, так как позволяет легко формировать динамические SQL-команды. Последний параметр метода представляет собой указатель на объект `TSQLCustomDataSet`. Этот объект создается в процессе вызова `Execute`, если сервер возвращает набор данных.

В качестве примера использования метода `Execute` рассмотрим процедуру `AddRecord` (листинг 14.4), добавляющую новую запись в таблицу `pricelist` базы данных `catalog` *(см. гл. 12)*.

Листинг 14.4 Процедура `AddRecord`

```
void TForm1::AddRecord (const String Name, const String Section,
const String Manufacturer, int Price, int Discount);
{
  TParams * cmdParams = new TParams;
  cmdParams->CreateParam(ftString, "Name", ptInput);
  cmdParams->CreateParam(ftString, "Sedction", ptInput);
  cmdParams->CreateParam(ftString, "Manufacturer", ptInput);
  cmdParams->CreateParam(ftInteger, "Price", ptInput);
  cmdParams->CreateParam(ftInteger, "Discount", ptInput);
  cmdParams->ParamByName("Name")->AsString = Name;
  cmdParams->ParamByName("Section")->AsString = Section;
  cmdParams->ParamByName("Manufacturer")->AsString = Manufacturer;
  cmdParams->ParamByName("Price")->AsInteger = Price;
  cmdParams->ParamByName("Discount")->AsInteger = Discount;
  String SQLCmd = "insert into pricelist values(:Name, :Section,↳
  :Manufacturer, :Price, :Discount)";
  SQLConnection1->Execute(SQLCmd, cmdParams, NULL);
  delete cmdParams;
}
```

Ряд методов компонента `SQLConnection` позволяет получать *метаданные* — информацию о различных объектах баз данных. Метод `GetTableNames` позволяет получить информацию о таблицах, определенных в базе данных. У данного метода два параметра: первый параметр — указатель на объект `TStrings`, в который записываются названия таблиц, второй параметр — переменная булевого типа, позволяющая указать, следует ли включать в список системные таблицы.

Еще один полезный метод — `GetFieldNames`, возвращающий список полей (названий столбцов) для указанной таблицы. У этого метода тоже два параметра. Первый параметр — строка с именем таблицы, второй параметр — указатель на объект `TStrings`, в который записываются названия полей.

В следующем фрагменте кода мы сначала получаем список всех несистемных таблиц базы данных, с которой установлена связь, и отображаем его в окне объекта `Memo1`, а затем, если этот список не пуст, перечисляем поля первой таблицы из полученного списка в окне объекта `Memo2`.

```
SQLConnection1->GetTableNames(Memo1->Lines, false);
if (Memo1->Lines->Count != 0)
SQLConnection1->GetFieldNames(Memo1->Lines->Strings[0], Memo2->Lines);
```

Компонент *SQLDataSet*

Компонент `SQLDataSet` представляет собой *однонаправленный набор данных* общего назначения.

К основным функциям компонента `SQLDataSet` относятся следующие:

- представление записей, содержащихся в таблице базы данных, возвращенных в результате выполнения SQL-команды `SELECT` или хранимой SQL-процедуры;
- выполнение команд и процедур языка SQL, не возвращающих данные для отображения;
- отображение метаданных, описывающих объекты, существующие в базе данных (таблицы, хранимые процедуры, типы полей таблиц и т. п.).

Так как компонент `SQLDataSet` является однонаправленным набором данных, это означает, что компонент не кэширует записи в памяти. В каждый данный момент времени компонент `SQLDataSet` "видит" только одну запись. По этой причине компонент `SQLDataSet` не позволяет использовать такие средства, как фильтры и поля просмотра. В данном компоненте также отсутствуют встроенные средства редактирования записей (хотя мы можем редактировать записи явным образом при помощи SQL-команд `INSERT` и `UPDATE`). Именно в этих особенностях компонента `SQLDataSet` заключается причина того, что ссылку на этот компонент нельзя использовать вместе с такими компонентами, как `DBGrid`.

Для того, чтобы компонент SQLDataSet мог обрабатывать данные, хранящиеся в БД, этот компонент должен быть связан с компонентом SQLConnection (при помощи свойства SQLConnection).

Редактирование запроса, направляемого компонентом SQLDataSet базе данных, может выполняться во время разработки программы при помощи редактора **CommandText Editor** (рис. 14.3). Для вызова этого редактора нужно в инспекторе щелкнуть кнопкой мыши на кнопке с символом многоточия в поле свойства CommandText. Редактор **CommandText Editor** — не просто текстовый редактор, позволяющий вводить SQL-запросы. Если компонент SQLDataSet связан с компонентом SQLConnection, который, в свою очередь, установил связь с базой данных (свойство Connected равно true), в окне **Tables:** редактора **CommandText Editor** будет отображен список таблиц базы данных, а в окне **Fields:** — список полей выбранной таблицы. Но это еще не все. Нажимая кнопки **Add Table to SQL** и **Add Field to SQL**, мы можем внести имена выбранных элементов в окно **SQL:**, где и выполняется редактирование запроса. После того как SQL-запрос отредактирован, нажмем кнопку **OK**.

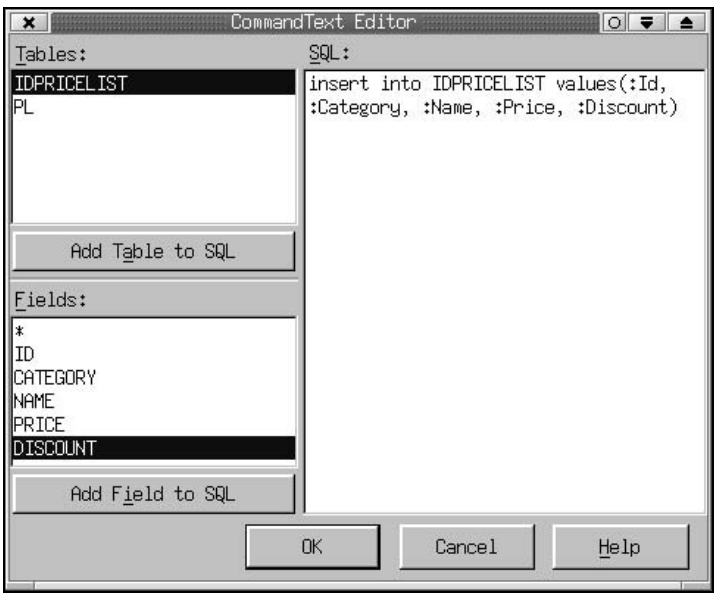


Рис. 14.3. Редактор **CommandText Editor**

Теперь мы можем назначить свойству Active компонента SQLDataSet значение true и проверить таким способом корректность созданного запроса к БД.

Свойство CommandText можно использовать и иначе. Раскрывающийся список свойства CommandType позволяет нам выбрать тип команды: ctQuery (по умолчанию), ctStoredProc и ctTable. Если мы выберем пункт ctTable, то

свойство CommandText изменит свой вид и превратится в список, в котором можно будет выбрать одну из таблиц базы данных. В этом случае фактический запрос, который компонент SQLDataSet отправит серверу БД, будет запросом на выдачу всех данных, содержащихся в выбранной таблице.

Весьма полезной возможностью компонента SQLDataSet является создание параметрических запросов. Для указания параметров запроса служит свойство Params типа TParams. Если мы щелкнем мышью на кнопке с символом многоточия в поле этого свойства, будет открыто окно редактора параметров (рис. 14.4). Этот редактор позволяет создавать новые объекты класса TParam, свойства которых можно редактировать в инспекторе объектов.

В качестве примера использования параметрических запросов напишем программу, переносящую в базу данных информацию из XML-документа. Пусть у нас имеется XML-документ, содержащий список товаров некоторого торгового предприятия в формате, показанном в листинге 14.5.

Листинг 14.5. XML-документ catalog.xml

```
<?xml version="1.0"?>
<catalog>
<item>
  <id>1</id>
  <category>Monitors</category>
  <name>Eye-nice(tm) Monitor</name>
  <price>400</price>
  <discount>10</discount>
</item>
<item>
  ...
</item>
...
</catalog>
```

Для удобства работы с XML-документом создадим для него интерфейсы C++ с помощью мастера **XML Data Bindings** *(см. разд. "Использование мастера XML Data Bindings" гл. 6)*. Сохраним полученные файлы catalog.cpp и catalog.h.

Создадим в выбранной нами СУБД новую базу данных, а в ней таблицу с помощью SQL-команды:

```
create table idpricelist(id int not null primary key, Category ↳
 varchar(10), Name varchar(32), Price int, Discount int);
```

Рис. 14.4. Редактор параметров компонента `SQLDataSet`

Создадим новый проект в Kylix C++ IDE, добавим в него компонент `SQLConnection` и настроим этот компонент на соединение с базой данных. Добавим в проект компонент `SQLDataSet` и свяжем его с компонентом `SQLConnection`.

Назначим свойству `ParamCheck` значение `false`. Вызовем редактор параметров компонента `SQLDataSet` (рис. 14.4) и создадим в нем новый параметр. Перейдем в инспектор объектов и назначим свойствам нового объекта `TParam` следующие значения: свойству `Name`, в котором хранится имя параметра — значение `Id`, свойству `DataType`, определяющему тип данных параметра — значение `ftInteger`, свойству `ParamType` присвоим значение `ptInput` (параметр для ввода значений). Аналогичным образом создадим еще четыре параметра, значения свойств которых приводятся в табл. 14.1.

Таблица 14.1. *Свойства параметров* `SQLDataSet`

Свойство	Значение
	Параметр 1
`Name`	`Category`
`DataType`	`ftString`
`ParamType`	`ptInput`
	Параметр 2
`Name`	`Name`
`DataType`	`ftString`
`ParamType`	`PtInput`

Таблица 14.1 (окончание)

Свойство	Значение
Параметр 3	
Name	Price
DataType	FtInteger
ParamType	PtInput
Параметр 4	
Name	Discount
DataType	ftInteger
ParamType	ptInput

С помощью редактора **CommandText Editor** зададим в свойстве `CommandText` следующий запрос:

```
insert into idpricelist values(:Id, :Category, :Name, :Price, :Discount)
```

Двоеточия в списке `values` означают, что следующие за ними выражения являются идентификаторами параметров. При выполнении запроса эти выражения будут заменены значениями соответствующих параметров.

Добавим в форму приложения компонент `OpenDialog` и стандартную кнопку. Назовем соответствующий объект `OpenButton`. Добавим в проект файл catalog.cpp, сгенерированный мастером **XML Data Bindings**, и вставим в файл исходного текста главного модуля строку:

```
#include "catalog.h"
```

Теперь назначим обработчик событию `OnClick` кнопки `OpenButton` (текст обработчика приводится в листинге 14.6).

Листинг 14.6. Обработчик события `OnClick` кнопки `OpenButton`

```
void __fastcall TForm1::OpenButtonClick(TObject *Sender)
{
  if (OpenDialog1->Execute())
  {
    _di_IXMLcatalogType Catalog;
    Catalog = Loadcatalog(OpenDialog1->FileName);
    for (int i = 0; i < Catalog->Count; i++)
```

```
    {
      _di_IXMLitemType Item = Catalog->item[i];
      SQLDataSet1->Params->ParamByName("Id")->AsInteger = Item->Get_id();
      SQLDataSet1->Params->ParamByName("Category")->AsString =
      (String)Item->Get_category();
      SQLDataSet1->Params->ParamByName("Name")->AsString =
      (String)Item->Get_name();
      SQLDataSet1->Params->ParamByName("Price")->AsInteger =
      Item->Get_price();
      SQLDataSet1->Params->ParamByName("Discount")->AsInteger =
      Item->Get_discount();
      Item->Release();
      SQLDataSet1->ExecSQL();
    }
    Catalog->Release();
  }
}
```

При нажатии на кнопку открывается диалоговое окно, в котором можно выбрать XML-файл, из которого будут импортироваться данные. Структура этого файла должна соответствовать структуре файла catalog.xml.

В обработчике события `OnClick` мы загружаем выбранный SQL-документ и затем, в цикле, заполняем параметры объекта `SQLDataSet1` данными, полученными из очередного элемента XML-документа, вызывая в конце каждой итерации метод `SQLDataSet1->ExecSQL()`, выполняющий сформированный SQL-запрос. Таким образом, все записи из XML-документа будут перенесены в таблицу `idpricelist`, в чем мы можем убедиться, выполнив в программе-клиенте СУБД SQL-команду:

```
select * from idpricelist;
```

Выше был описан процесс создания параметров явным образом. Если мы хотим, чтобы созданный нами список параметров `TParams` сохранялся при редактировании запросов, установим значение свойства `ParamCheck`, равным `false`. Если свойство `ParamCheck` равно `true`, список параметров `Params` автоматически регенерируется после изменения запроса. Это сделано для удобства программиста. Внесем в форму приложения новый компонент `SQLDataSet`, свяжем новый объект с объектом `SQLConnection1` и укажем следующий запрос для объекта `SQLDataSet2`:

```
select * from idpricelist where category = :Category
```

После этого откроем редактор параметров объекта `SQLDataSet2` и увидим автоматически созданный параметр `Category`, которому уже присвоен "правильный" тип `ftString`. Таким образом, мы можем создавать пара-

метры запроса, просто указывая значения, предваренные двоеточием в тексте SQL-запроса.

Многие параметры запроса, которые могут быть указаны явным образом в свойстве `CommandText`, можно указать при помощи редакторов свойств компонента `SQLDataSet`. Свойство `SortFieldNames` позволяет указывать порядок сортировки данных. Редактор этого свойства показан на рис. 14.5.

Список **Available Fields** представляет все поля таблицы, а список **Order by Fields** — список полей, по которым выполняется сортировка.

Выбранный порядок полей означает, что первичная сортировка выполняется в соответствии со значениями поля `price`, а вторичная (для записей с одинаковым значением — `price`) - по значениям поля `name`.

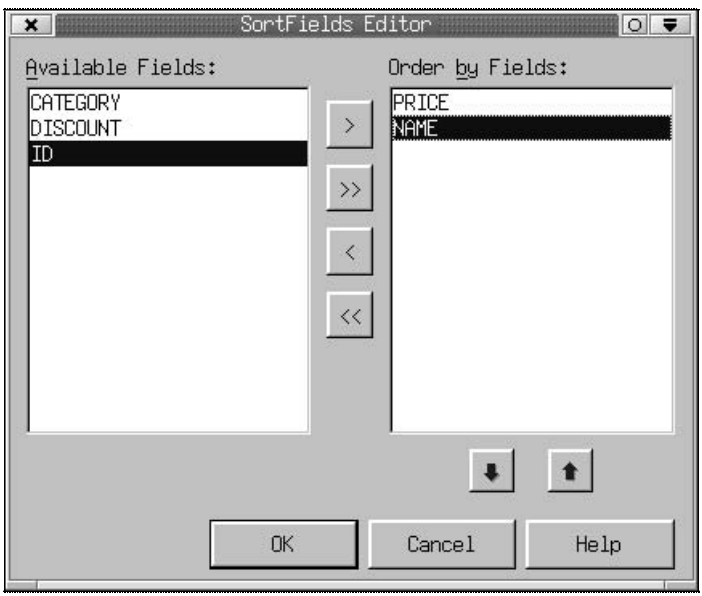


Рис. 14.5. Редактор свойства `SortFieldNames`

Компонент *SQLQuery*

Компонент `SQLQuery`, как и компоненты `SQLTable` и `SQLStoredProc`, можно рассматривать как "облегченные" версии компонента `SQLDataSet`. Компонент `SQLDataSet` включает в себя все возможности этих компонентов. Как и компонент `SQLDataSet`, все эти компоненты представляют собой однонаправленный набор данных и позволяют направлять серверу БД SQL-команды. Для указания SQL-команд служит свойство `SQL`, представляющее собой обычный набор строк. Поддерживаются параметрические запросы и

механизм автоматической генерации параметров на основе текста запроса. Для передачи серверу БД запроса, сгенерированного во время выполнения программы, можно использовать метод `ExecSQL`.

Компонент *SQLStoredProc*

Компонент предназначен для вызова хранимой процедуры (stored procedure). Имя процедуры задается в свойстве `StoredProcName`. Поддерживается использование параметров. Вызов хранимой процедуры осуществляется при помощи метода `ExecProc`.

Компонент *SQLTable*

Компонент `SQLTable` предназначен для доступа к отдельным таблицам базы данных. Свойство `TableName` позволяет выбрать таблицу.

Компонент *SQLMonitor*

Этот компонент предназначен для ведения журнала событий, происходящих при обращении к базе данных. Возможны два режима записи информации о событиях. В первом случае вся информация автоматически записывается в файл, хранящийся на диске. Для этого имя файла должно быть указано в свойстве `FileName`, а свойство `AutoSave` должно быть установлено равным `true`. При втором варианте все события записываются в компонент-коллекцию строк (свойство `TraceList`), а сохранить запись можно при помощи метода `SaveToFile`, которому передается имя файла. Если мы хотим просматривать или изменять сообщения, помещаемые в журнал событий во время исполнения программы, назначим обработчик событию `OnTrace`. Обработчику передается переменная `CBInfo`, содержащая информацию, которая должна быть занесена в журнал событий. С помощью булевого параметра-переменной `LogTrace` мы можем указать, следует ли заносить данную информацию в журнал или нет. Другое событие `OnLogTrace` — вызывается сразу после занесения очередной записи в журнал, но перед тем, как запись будет сохранена на диске (если выбран режим автоматического сохранения).

Подключить компонент `SQLMonitor` к отслеживаемому соединению с базой данных очень просто. Для этого нужно назначить свойству `SQLConnection` ссылку на соответствующий объект `SQLConnection` и установить значение свойства `Active` объекта `SQLMonitor`, равным `true`.

Компонент *SQLClientDataSet*

Компонент `SQLClientDataSet` можно рассматривать как гибрид компонентов `SQLDataSet` и `ClientDataSet`. Компонент `SQLClientDataSet` является

клиентским набором данных и включает в себя компоненты `SQLDataSet` и `DataSetProvider`. Таким образом, применение этого компонента избавляет программиста от необходимости использовать явным образом как компонент `SQLDataSet`, так и компонент-провайдер. Будучи клиентским набором данных, компонент `SQLClientDataSet` кэширует записи в памяти и позволяет вносить в них изменения с помощью графических элементов управления базами данных таких, как `DBGrid`.

Компонент `SQLClientDataSet` позволяет управлять свойствами, унаследованными от классов `TSQLDataSet` и `TClientDataSet`. Если наше приложение является dbExpress-специфичным, мы можем выбрать компонент `SQLClientDataSet` в качестве его основы.

Как и все клиентские наборы данных, компонент `SQLClientDataSet` позволяет сохранять данные из своего набора в файле на диске. Эту операцию выполняет метод `SaveToFile`. Методу `SaveToFile` передаются два параметра: имя файла, в котором следует сохранять данные, и формат данных. Поддерживаются три формата: двоичный, XML и XML в кодировке UTF8.

Компонент `SQLClientDataSet` может также загружать данные из файла, в котором они были ранее сохранены в одном из указанных форматов. Во время выполнения программы это можно сделать, например, с помощью метода `LoadFromFile`, а на этапе разработки присвоить имя соответствующего файла свойству `FileName`, видимому в инспекторе объектов. В этом случае мы можем работать с набором данных, не устанавливая связь с какой-либо СУБД (то есть без помощи компонента `SQLConnection`). Роль базы данных будет выполнять локальный файл. При этом все графические элементы управления базами данных будут работать так же, как и в случае "настоящей" БД. Сохранение наборов данных в локальных файлах можно использовать и для переноса данных между различными СУБД, и для аварийного сохранения внесенных изменений в случае внезапного разрыва связи в распределенной системе, и для отладки других компонентов приложения при отсутствии связи с сервером баз данных.

Более подробно о свойствах и методах компонента `SQLClientDataSet`, общих с другими клиентскими наборами данных, но можем прочесть в следующей главе в разделе, посвященном компоненту `ClientDataSet`.

Создание приложения просмотра баз данных на основе компонентов dbExpress

В предыдущих разделах этой главы приводились фрагменты исходных текстов, демонстрирующие различные возможности компонентов dbExpress. В этом разделе мы напишем законченную программу (пусть и достаточно простую), использующую компоненты dbExpress. Эта программа не просто

демонстрирует работу с ранее описанными компонентами. В ней будут задействованы некоторые возможности dbExpress, которых мы еще не касались.

В качестве основы приложения мы используем базу данных, хранящую таблицу `idpricelist`, созданную в предыдущем разделе. Наше приложение позволит пользователю не только просматривать содержимое таблицы целиком, но и выбирать из нее отдельные подмножества записей на основе заданных им критериев. Проще говоря, поскольку таблица `idpricelist` представляет собой каталог товаров, наше приложение позволит пользователю выбирать записи об отдельных товарах на основании таких параметров, как категория товара и цена.

Для того чтобы получить информацию из базы данных, пользователь должен указать интересующую его категорию товаров и выбрать нижнюю и верхнюю границы ценового диапазона. Поскольку категорий товаров в каталоге не так уж и много, удобно реализовать выбор необходимой категории с помощью раскрывающегося списка, причем желательно, чтобы список содержал только те категории, которые фактически присутствуют в каталоге. Можно ввести и еще одно удобство: при выборе категории товара поля, предназначенные для ввода границ ценового диапазона, должны автоматически заполняться соответственно минимальным и максимальным значением цены для товаров данной категории, представленных в каталоге.

Создадим новый проект в Kylix C++ IDE и разместим в нем компонент `SQLConnection`, настроив его на связь с базой данных, содержащей таблицу `idpricelist`. Поместим в форму два компонента `SQLDataSet`, компонент `SQLClientDataSet`, компонент `DataSource` и компонент `DBGrid`. Эти компоненты составят ту часть приложения, которая связана с базой данных.

Назначим свойству `DataSource` объекта `DBGrid1` ссылку на объект `DataSource1`, а свойству `DataSet` этого объекта — ссылку на объект `SQLClientDataSet1`. Можно догадаться, что свойству `DBConection` объекта `SQLClientDataSet1` следует присвоить ссылку на объект `SQLConnection1`. На этот же объект должны ссылаться свойства `SQLConnection` объектов `SQLDataSet1` и `SQLDataSet2`. Как мы увидим далее, в принципе, нам было бы достаточно одного компонента `SQLDataSet`, но для упрощения примера мы используем два.

Возможно, описание взаимосвязей между компонентами покажется слишком сложным. В любом случае это хороший повод продемонстрировать новую возможность Kylix 3 — генерацию графических схем взаимосвязей объектов. Схема для вышеперечисленных объектов показана на рис. 14.6. Для того, чтобы сгенерировать такую схему, необходимо открыть страницу **Diagram** редактора кода и "перетащить" в нее нужные объекты из окна **Object Tree View**. Надписи на стрелках — названия свойств, осуществляющих связи.

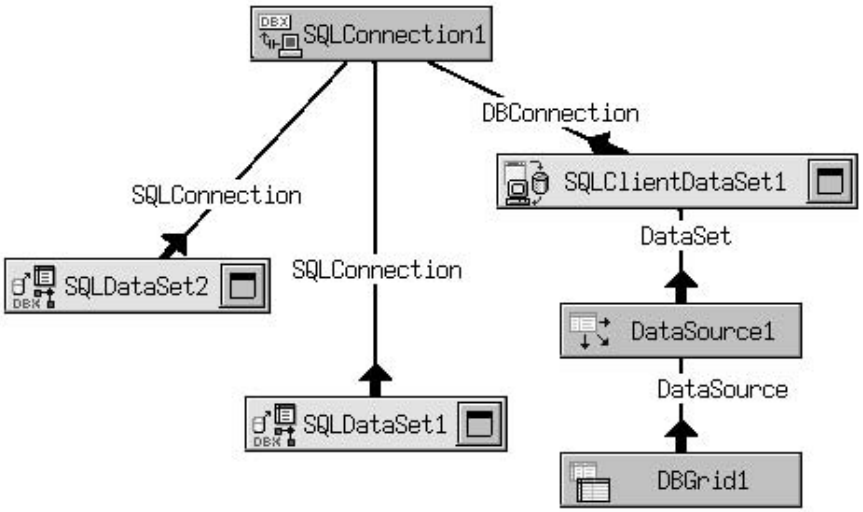


Рис. 14.6. Схема связей между объектами доступа к базе данных

Свойству `CommandType` объекта `SQLClientDataSet1` присвоим `ctTable` и в раскрывающемся списке, которым станет свойство `CommandText`, выберем значение `idpricelist`. Таким образом, мы настраиваем объект `SQLClient-DataSet1` на просмотр таблицы. Установим значение свойства `Filtered` объекта `SQLClientDataSet1`, равным `true`, и это же значение присвоим свойству `foCaseInsensitive` из группы свойств `FilterOptions`. Эти операции необходимы для установки *фильтра* для объекта `SQLClientDataSet1`. Сами фильтры будут устанавливаться во время работы приложения.

Теперь создадим раскрывающийся список для выбора категории товара. В качестве визуального элемента управления используем компонент `ComboBox` (разместите этот компонент в форме и присвоим соответствующему объекту имя `CategoryComboBox`), в качестве же источника данных для списка мы используем объект `SQLDataSet1`. В редакторе SQL-запросов этого объекта введем запрос:

```
select distinct category from idpricelist
```

Этот запрос возвращает таблицу, содержащую значения столбца `category` таблицы `idpricelist`. Указание `distinct` необходимо потому, что в таблице `idpricelist` одно и то же наименование категории товара может встречаться сколь угодно много раз.

В таблице `idpricelist` наименования категорий товаров следуют в произвольном порядке. В таком же порядке они будут возвращены и в таблице,

полученной в результате приведенного выше запроса. Если мы хотим, чтобы в раскрывающемся списке наименования категорий были отсортированы, мы можем добавить указание `order by category` в текст SQL-запроса, а можно просто присвоить значение `true` свойству `Sorted` объекта `CategoryComboBox`. Так как список категорий товаров не допускает редактирования и ввода произвольных значений, свойству `Style` объекта `CategoryComboBox` следует присвоить значение `csDropDownList`.

Напишем функцию `FillCombo` (эта функция должна быть методом класса главной формы), заполняющую список `CategoryComboBox` значениями категорий товаров. Текст этой функции приводится в листинге 14.7.

Листинг 14.7. Функция `FillCombo`

```
void TForm1::FillCombo()
{
  TField * Field;
  SQLDataSet1->Open();
  while (!SQLDataSet1->Eof)
  {
    Field = SQLDataSet1->FieldByName("category");
    String FV = Field->AsString;
    CategoryComboBox->Items->Add(FV);
    SQLDataSet1->Next();
  }
  SQLDataSet1->Close();
  if (CategoryComboBox->Items->Count > 0)
  CategoryComboBox->Text =
  CategoryComboBox->Items->Strings[CategoryComboBox->Items->Count-1];
}
```

В функции `FillCombo` используются методы и свойства класса `TSQLDataSet`, которые мы не рассматривали раньше. Прежде всего, это метод `Open`, открывающий набор данных. Вызов метода `Open` аналогичен присвоению значения `true` свойству `Active`.

Компоненты, реализующие наборы данных, как однонаправленные, так и клиентские, позволяют работать с отдельными записями таблицы. Запись представляется совокупностью полей, соответствующих столбцам таблицы, которые можно извлекать различными способами. В функции `FillCombo` используется метод `FieldByName`, возвращающий поле с указанным именем для текущей (активной) записи таблицы. Метод `FieldByName` возвращает указатель на объект класса `TField`. Свойство `DataType` этого объекта позво-

ляет получить сведения о типе данных, к которому принадлежит значение поля, а группа свойств, начинающихся с префикса "As", позволяет получить значение поля в одном из поддерживаемых типов (который не обязательно должен совпадать со значением свойства `DataType`). При этом, если необходимо, может быть выполнено преобразование типов. Для присвоения значений полю записи можно также использовать свойство `Value` вариантного типа. Еще одно интересное свойство класса `TField` — свойство `EditMask`, позволяющее устанавливать для поля маску ввода, аналогичную маске ввода компонента `MaskedEdit`. К каким ячейкам будет применена эта маска? Ко всем ячейкам столбца. В этом заключается определенная двойственность объектов `TField`. С одной стороны, такие свойства этих объектов, как `AsString` или `Value`, позволяют обращаться к отдельной ячейке столбца, соответствующей текущей записи (строке) таблицы. С другой стороны, такие свойства, как `DataType`, `FieldName` и `EditMask`, определяют параметры всего столбца.

Однонаправленные наборы данных позволяют просматривать записи только последовательно, в прямом или обратном направлении. Для перехода к следующей записи служит метод `Next` объекта-потомка `TDataSet`, а для перехода к предыдущей — метод `Prior`. Установить текущую позицию на первой записи можно с помощью метода `First` (при открытии набора данных первая запись является текущей по умолчанию), а метод `Last` позволяет сделать текущей последнюю запись. Свойство `Eof` принимает значение `true`, когда достигнута последняя запись, а метод `Bof` — когда достигнута первая (это удобно при просмотре записей в обратном порядке). Закончив работу с набором данных, мы закрываем его с помощью метода `Close`.

Для указания границ ценового диапазона мы введем два компонента `Edit` (назовем объекты соответственно `MinPriceEdit` и `MaxPriceEdit`). Мы хотим, чтобы для удобства пользователя при выборе категории товара эти элементы принимали значения, соответствующие минимальному и максимальному значению цены для товаров данной категории (пользователь может затем изменить границы ценового диапазона по своему желанию).

Для того чтобы получить минимальное и максимальное значение цены для данной категории товаров, мы воспользуемся объектом `SQLDataSet2`. Назначим этому объекту SQL-запрос:

```
select min(price), max(price) from idpricelist where category = :Category
```

Как видим, запрос параметрический, и параметр `Category` будет создан для него автоматически. Результатом выполнения запроса должна стать таблица из двух столбцов, первый из которых будет содержать минимальное значение цены для данной категории, а второй — максимальное.

Рассмотрим функцию `FillEdits` (листинг 14.8), считывающую значения из таблицы и заполняющую ими объекты `MinPriceEdit` и `MaxPriceEdit`.

Листинг 14.8. Функция `FillEdits`

```
void TForm1::FillEdits()
{
  if (CategoryComboBox->Text!="")
  {
    TField * Field;
    SQLDataSet2->Params->ParamByName("Category")->AsString =
   (String)CategoryComboBox->Text;
    SQLDataSet2->Open();
    Field = SQLDataSet2->Fields->Fields[0];
    MinPriceEdit->Text = Field->AsString;
    Field = SQLDataSet2->Fields->Fields[1];
    MaxPriceEdit->Text = Field->AsString;
    SQLDataSet2->Close();
  }
}
```

В этой функции мы сначала устанавливаем значение параметра для запроса. При вызове метода `Open` запрос выполняется автоматически с указанным значением параметра `Category`. Следует подчеркнуть, что в функции `FillEdits` мы используем другой метод обращения к полям записи. Дело в том, что поля записи (столбцы таблицы) максимальных и минимальных значений могут иметь разные названия в различных СУБД. Для того, чтобы наше демонстрационное приложение могло работать с СУБД различных типов, мы обращаемся к полям не по именам, а по индексам.

Все элементы ввода приложения должны быть заполнены корректными значениями сразу после запуска. Обработчик события `OnShow` главной формы представлен в листинге 14.9.

Листинг 14.9. Обработчик события `OnShow`

```
void __fastcall TForm1::FormShow(TObject *Sender)
{
  FillCombo();
  FillEdits();
}
```

Для того чтобы значения цен в полях ввода диапазона менялись при выборе категории, следует назначить очень простой обработчик события `OnSelect` объекта `CategoryComboBox` (листинг 14.10).

Листинг 14.10. Обработчик события `OnSelect` объекта `CategoryComboBox`

```
void __fastcall TForm1::CategoryComboBoxSelect(TObject *Sender)
{
  FillEdits();
}
```

Для того чтобы после выбора категории товара и ценового диапазона пользователь мог направить запрос к базе данных, мы используем компонент `Button` (назовем объект `ShowQueryButton`) и назначим его событию `OnClick` обработчик, представленный в листинге 14.11.

Листинг 14.11. Обработчик события `OnClick` объекта `ShowQueryButton`

```
void __fastcall TForm1::ShowQueryButtonClick(TObject *Sender)
{
  String FS, Category, MinPrice, MaxPrice;
  Category = CategoryComboBox->Text;
  MinPrice = MinPriceEdit->Text;
  MaxPrice = MaxPriceEdit->Text;
  FS.printf("category = '%s' and price >= %s and price <= %s",
  &Category[1], &MinPrice[1], &MaxPrice[1]);
  SQLClientDataSet1->Filter = FS;
  SQLClientDataSet1->Refresh();
}
```

Выше уже отмечалось, что функции компонентов dbExpress дублируют функции языка SQL.

Примечание

Впрочем, дублирование возникает только в случае взаимодействия с SQL-СУБД. В следующей главе мы увидим, что при использовании других источников данных эти "дублирующие" функции оказываются весьма полезными.

К дублирующим функциям относятся уже упоминавшиеся фильтры, реализованные в клиентских наборах данных, в том числе и компоненте `SQLClientDataSet`. Фильтр представляет собой строку, содержащую выражение, напоминающее выражение критерия выбора языка SQL. Если в объекте, реализующем клиентский набор данных, установлен фильтр, в результирующий набор войдут только те записи, которые соответствуют критериям фильтра. Мы уже указали, что клиентский набор данных `SQLClientDataSet1` должен использовать фильтры. Теперь мы устанавливаем фильтр, соответствующий запросу пользователя. В обработчике события в

листинге 14.11 выражение для фильтра формируется в строке `FS` с помощью метода `printf`. Сформировав критерий, мы присваиваем значение строки `FS` свойству `Filter` объекта `SQLClientDataSet1`. После этого нужно вызвать метод `Refresh`, обновляющий содержимое набора данных в соответствии с вновь сформированным запросом.

Для удобства можно добавить кнопку для отображения всей таблицы. Для того, чтобы таблица отображалась целиком, следует присвоить пустую строку свойству `Filter` набора данных и вызвать метод `Refresh`. Работающее приложение показано на рис. 14.7.

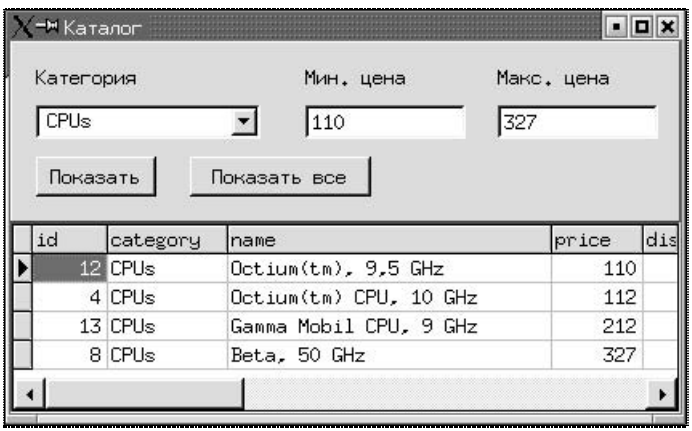


Рис 14.7. Приложение просмотра базы данных

Состояния набора данных

Свойство `State`, предназначенное только для чтения, позволяет определить текущее состояние набора данных. Состояния набора данных обозначаются специальными константами, являющимися элементами перечислимого типа `TDataSetState`. Возможными состояниями набора данных являются `dsBrowse` — состояние просмотра, `dsEdit` — состояние редактирования текущей записи, `dsInsert` — состояние вставки записи и многие другие. Мы не можем изменить состояние набора данных явным образом. Состояние изменяется соответствующими методами объекта, реализующего набор данных. Множество возможных состояний для клиентского набора данных шире множества возможных состояний однонаправленного набора.

Глава 15

Локальные приложения баз данных

В этой главе мы рассмотрим утилиты и компоненты, предоставляемые средой Borland Kylix для разработки локальных приложений баз данных. Даже если мы собираемся создавать исключительно распределенные и интернет-приложения, все равно следует прочесть эту главу, так как описанные в ней компоненты могут быть полезны, а иногда необходимы, при разработке приложений других архитектур.

Клиентские наборы данных и компоненты-провайдеры

В этом разделе речь пойдет, прежде всего, о компоненте `ClientDataSet`, который может использоваться как в приложениях, работающих с SQL-базами данных, так и в приложениях других типов. Как уже неоднократно отмечалось, особенность клиентских наборов данных состоит в том, что компоненты, реализующие эти наборы данных, хранят в памяти набор записей. Этот набор может содержать все записи таблицы, с которой работает компонент, либо некоторое подмножество записей. В пределах хранящегося в памяти набора записей возможна быстрая (и произвольная) навигация. Эта возможность также является отличительной чертой клиентских наборов данных.

Фактически компонент `ClientDataSet` содержит в памяти не один, а два набора записей. Первый носит название `Data`. Он отражает текущее состояние набора данных. Второй набор записей называется `Delta`. Он содержит информацию обо всех изменениях, произведенных в наборе данных. Когда в клиентский набор данных вносятся изменения, они тут же отображаются в наборе записей `Data`, а в набор записей `Delta` вносится полная информация о внесенных изменениях. При этом никаких изменений в первичном хранилище данных еще не производится. Благодаря набору `Delta` мы можем отменить внесенные изменения. Этот набор используется также при внесении изменений в первичный источник данных. При открытии клиентского набо-

ра данных набор записей Data получает точную копию таблицы (или ее фрагмента) от компонента-провайдера *(см. гл. 11)*. В этот момент набор Delta не содержит никаких записей. В нем отображаются все изменения, которые затем вносятся в набор данных. Когда программа вносит сделанные изменения в первичный источник данных (с помощью метода ApplyUpdates), операции по изменению содержимого первичного источника производятся на основе информации, хранящейся в наборе Delta. Этот набор передается компоненту-провайдеру, который при помощи довольно сложного механизма формирует последовательность SQL-команд, вносящих изменения в базу данных в соответствии с содержимым набора Delta. После внесения изменений в первичный источник данных этот набор снова становится пустым.

Клиентские наборы данных отличаются не только возможностью быстрой навигации по записям таблицы. Важной их особенностью является также возможность сохранять наборы данных в файле на диске. Эта возможность позволяет приложениям, использующим клиентские наборы данных, работать в автономном режиме в случае отсутствия связи с сервером баз данных. При сохранении набора данных в локальном файле сохраняется не только набор записей Data, но и набор Delta, что позволяет позднее, при подключении к серверу, внести в базу данных изменения, сделанные при работе в автономном режиме. Поскольку эта возможность играет большую роль в распределенных приложениях баз данных, нежели в локальных, подробнее она будет рассмотрена в следующей главе.

Впрочем, сохранение наборов данных на диске может использоваться не только в приложениях, предполагающих "воссоединение" с сервером баз данных. Поскольку клиентские наборы данных взаимодействуют с другими элементами приложения одинаково, независимо от того, получают ли они пакеты данных от компонентов-провайдеров или загружают их из локальных файлов, мы можем построить приложение, имеющее архитектуру, подобную архитектуре приложений БД, но работающее исключительно с локальными файлами. В документации Borland эта схема работы носит название MyBase.

Продемонстрируем использование схемы MyBase на примере, в котором также затронем еще один интересный момент — создание таблицы с помощью компонент-клиентского набора данных.

В новый проект приложения на языке C++ добавим компоненты ClientDataSet, DataSource, DBGrid и DBNavigator. Поскольку создаваемое приложение будет работать исключительно с локальными файлами, ни компоненты-провайдеры, ни компоненты, обеспечивающие связь с базой данных, нам не понадобятся. Свяжем свойства DataSource объектов DBNavigator1 и DBGrid1 с объектом DataSource1, а свойство DataSet объекта DataSource1 с объектом ClientDataSet1.

Добавим в проект три компонента-кнопки BitBtn. Первая кнопка (назовем соответствующий объект CreateBitBtn) отвечает за создание новой табли-

цы. Вторая кнопка (SaveBitBtn) служит для сохранения таблицы в файле, а третья (OpenBitBtn) — за загрузку таблицы из файла. Текст обработчиков событий приводится в листинге 15.1.

Листинг 15.1. Обработчики событий CreateBitBtn, SaveBitBtn и OpenBitBtn

```
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
  OpenDialog1->Filter = "жБКМЩ cds | *.cds | жБКМЩ xml | *.xml";
  SaveDialog1->DefaultExt = ".cds";
}
//---------------------------------------------------------------------------

void __fastcall TForm1::CreateBitBtnClick(TObject *Sender)
{
  if (ClientDataSet1->FieldDefs->Count == 0)
  {
    TFieldDef * NewField;
    NewField = ClientDataSet1->FieldDefs->AddFieldDef();
    NewField->Name = "id";
    NewField->DataType = ftInteger;
    NewField->Attributes << faRequired;
    NewField = ClientDataSet1->FieldDefs->AddFieldDef();
    NewField->Name = "category";
    NewField->DataType = ftString;
    NewField->Size = 10;
    NewField = ClientDataSet1->FieldDefs->AddFieldDef();
    NewField->Name = "name";
    NewField->DataType = ftString;
    NewField->Size = 32;
    NewField = ClientDataSet1->FieldDefs->AddFieldDef();
    NewField->Name = "price";
    NewField->DataType = ftInteger;
    NewField = ClientDataSet1->FieldDefs->AddFieldDef();
    NewField->Name = "discount";
    NewField->DataType = ftInteger;
  }
```

```
  else
  ClientDataSet1->Close();
  ClientDataSet1->CreateDataSet();
}
//---------------------------------------------------------------------
void __fastcall TForm1::SaveBitBtnClick(TObject *Sender)
{
  if (SaveDialog1->Execute())
  {
    String Ext = ExtractFileExt((String)SaveDialog1->FileName);
    if (Ext == ".cds")
    ClientDataSet1->SaveToFile((String)SaveDialog1->FileName);
    else
    ClientDataSet1->SaveToFile((String)SaveDialog1->FileName, dfXML);
  }
}
//---------------------------------------------------------------------
void __fastcall TForm1::OpenBitBtnClick(TObject *Sender)
{
  if (OpenDialog1->Execute())
  ClientDataSet1->LoadFromFile((String)OpenDialog1->FileName);
}
//---------------------------------------------------------------------
```

В обработчике `CreateBitBtnClick` мы сначала проверяем, не создана ли уже таблица. Делается это при помощи обращения к свойству `Count` свойства `FieldDefs` объекта `ClientDataSet1`. Свойство `FieldDefs` содержит описание всех полей (столбцов) таблицы и, если значение свойства `Count` этого списка не равно нулю, значит, таблица уже существует. Следует отметить, что структура, описывающая таблицу, существует независимо от того, открыт набор данных или закрыт, и не уничтожается при закрытии набора данных.

Если структуры таблицы еще нет, мы создаем ее с помощью метода `AddFieldDef` списка `FieldDefs`. Данный метод возвращает указатель на созданный объект класса `TFieldDef`, описывающий свойства нового поля таблицы. Среди свойств класса `TFieldDef` особого внимания заслуживают следующие:

- `FieldNo` — указывает порядковый номер столбца в таблице;
- `Name` — указывает название столбца;
- `DataType` — указывает тип данных столбца;
- `Size` — указывает размер данных столбца (для типов переменного размера);

- `Attributes` — хранит список атрибутов столбца;
- `Required` — указывает, может ли поле быть пустым.

Свойство `Required` дублирует одно из значений свойства `Attributes`, но обращаться к нему, как правило, более удобно, нежели к свойству `Attributes`.

В обработчике `CreateBitBtnClick` мы последовательно заполняем свойства каждого созданного объекта класса `TFieldDef` и, таким образом, создаем структуру таблицы. Если бы объект `ClientDataSet1` был связан с объектом-провайдером, структура таблицы была бы передана объекту `ClientDataSet1` провайдером. В этом случае список `FieldDefs` можно было бы использовать для получения сведений о столбцах таблицы. После того, как таблица создана, необходимо создать набор данных. Так как объект `ClientDataSet1` не связан ни с каким провайдером, нельзя открыть набор данных с помощью метода `Open`. Набор данных создается автоматически при открытии соответствующего файла. Однако, когда мы создаем таблицу, у нас может не быть такого файла (ведь именно для создания файлов наборов данных и создается таблица). Поэтому мы создаем набор данных явно при помощи метода `CreateDataSet`, который создает пустой набор записей для таблицы с указанной структурой.

Обработчик `OpenBitBtnClick` загружает файл с диска при помощи метода `LoadFromFile`. При этом прежний набор записей (если он существовал) уничтожается и создается новый набор, соответствующий содержимому открываемого файла.

Обработчик `SaveBitBtnClick` сохраняет текущий набор записей в файле на диске при помощи метода `SaveToFile`. Если бы имя файла для сохранения данных было назначено свойству `FileName` объекта `ClientDataSet1`, данные сохранялись бы автоматически при каждом вызове метода `Close`. Метод `SaveToFile` позволяет сохранять наборы записей в различных форматах (двоичный и XML).

Работающее приложение показано на рис. 15.1.

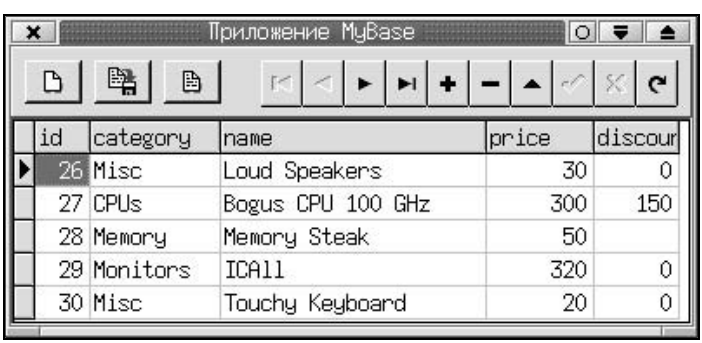


Рис. 15.1. Работающее приложение MyBase

Работа с областями (Ranges)

Клиентские наборы данных хранят в памяти наборы записей, полученных из таблиц (как правило, в памяти хранятся не один, а два набора). С этим связана одна проблема, которая заключается в том, что количество записей в таблице может быть очень большим. Для упрощения работы с таблицами, содержащими большое число записей, служат *области* (ranges). Область представляет собой непрерывную последовательность записей таблицы, являющуюся подмножеством множества всех записей. Следует понимать разницу между наборами записей, сформированными в результате применения фильтров, и областями. Наборы, сформированные при помощи фильтров, представляют собой совокупности записей, выбранных из всей таблицы на основе определенного критерия, в то время как области представляют собой фрагменты таблицы, заданные начальной и конечной записью. Рассмотрим, как устанавливается и применяется область (листинг 15.2).

Листинг 15.2. Определение области

```
SQLClientDataSet1->SetRangeStart();
SQLClientDataSet1->FieldByName("id")->AsInteger = 10;
SQLClientDataSet1->SetRangeEnd();
SQLClientDataSet1->FieldByName("id")->AsInteger = 20;
SQLClientDataSet1->ApplyRange();
```

В этом примере мы устанавливаем область для объекта класса `TSQLClientDataSet`, который, напомним, также является клиентским набором данных. Метод `SetRangeStart`, унаследованный от класса `TCustomClientDataSet`, позволяет указать начало области. Этот метод переводит набор данных в состояние ожидания ввода значения, идентифицирующего начальную запись области. Это значение мы вводим с помощью метода `FieldByName`. Конец области устанавливается с помощью метода `SetRangeEnd()`. После этого мы снова вызываем метод `FieldByName`, в котором устанавливаем значение поля последней записи области. Таким образом, мы определили область, содержащую записи со значениями поля `id`, начиная с 10 и заканчивая 20. Для указания области мы могли бы использовать значения и других полей, более того, для указания начала области можно использовать одно поле, а для указания конца — другое.

Когда границы области определены, следует вызвать метод `ApplyRange`. После вызова этого метода набор данных будет содержать только записи, входящие в выделенную область. Все описанные выше действия должны выполняться с открытым набором данных.

Мы можем редактировать записи, входящие в область, а также добавлять новые. Если мы добавляем новую запись, эта запись будет добавлена в на-

бор данных независимо от того, входит ли она в выделенную область или нет. После вызова метода `ApplyUpdates` эта запись будет внесена в базу данных, но в наборе данных новая запись будет присутствовать только в том случае, если она входит в выделенную область.

Что значит "входит в выделенную область"? Обычно для выделения области мы используем неповторяющиеся значения полей, по которым может выполняться сортировка. Если новая запись входит в выделенный диапазон, она будет присутствовать в наборе данных и после вызова метода `ApplyUpdates`. Например, если при работе с определенной выше областью мы добавим запись со значением поля `id`, равным 15, эта запись останется в наборе данных, а если значение поля `id` новой записи будет равно, например, 21, эта запись будет внесена в базу данных после вызова `ApplyUpdates`, но из набора данных она будет удалена.

Индексы

Индексы представляют собой объекты наборов данных, позволяющие осуществлять сортировку и поиск записей по значениям различных полей. Индексы клиентских наборов данных подобны индексам SQL, однако совершенно независимы от них. Если в SQL-базе данных и определены индексы, связанные с какой-либо таблицей, клиентский набор, использующий эту таблицу, ничего о них не знает. Если мы хотим использовать индексы в клиентских наборах данных, мы должны создать их явным образом. В клиентских наборах данных индексы, во-первых, ускоряют поиск записей с помощью описанных выше методов, если в критериях поиска используются индексированные поля, а, во-вторых, позволяют сортировать записи по значениям полей.

По умолчанию клиентский набор данных создает два индекса: `DEFAULT_ORDER` и `CHANGEINDEX`. Индекс `DEFAULT_ORDER` представляет порядок, в котором записи были получены от компонента-провайдера. Если в SQL-запросе содержалось указание `ORDER BY`, порядок записей по умолчанию будет соответствовать порядку, заданному этим указанием. Индекс `CHANGEINDEX` отражает порядок записей в пакете данных `Delta`.

Мы можем создавать для набора данных дополнительные индексы. Существует два способа создания индексов: с помощью свойства `IndexFieldNames` и с помощью свойства `IndexDefs`. Для создания индекса с помощью свойства `IndexFieldNames` достаточно присвоить этому свойству строку, содержащую список индексируемых полей, разделенных символом ";". Созданные таким образом индексы позволяют сортировать записи по значениям индексируемых полей в порядке возрастания. При этом первичная сортировка производится по значениям первого поля в списке, вторичная — по значениям второго поля и т. п. Например, присваивание:

```
ClientDataSet1->IndexFieldNames = "CATEGORY;PRICE";
```

приведет к созданию нового индекса. Записи таблицы будут отсортированы по значениям поля `CATEGORY`, а для одинаковых значений этого поля —

дополнительно по значениям поля PRICE. Сортировка записей по указанным полям будет выполнена сразу после выполнения операции присваивания.

Достоинство подобных индексов — простота их создания. Этот тип индексов удобно применять в тех случаях, когда требуется предоставить пользователю возможность сортировки записей по произвольно выбранным полям. Недостатком таких индексов является невозможность определения дополнительных свойств и изменения порядка сортировки.

Индексы, создаваемые с помощью свойства IndexDefs, обладают гораздо большими возможностями. Свойство IndexDefs представляет собой список объектов класса TIndexDef, каждый из которых описывает один индекс. Индексы этого типа позволяют сортировать записи не только в порядке возрастания, но и убывания, а также позволяют определить ряд дополнительных свойств индекса. Объекты TIndexDef можно создавать как во время разработки программы, так и во время выполнения. Для того, чтобы создать индексы во время разработки, щелкним мышью на кнопке с символом многоточия в поле свойства IndexDefs. При этом открывается типичное окно редактора списка. Создав новый объект класса TIndexDef в инспекторе объектов, можно присвоить значения основным свойствам объекта. Среди этих свойств выделим следующие:

- Name — название, идентифицирующее индекс;
- Fields — список индексируемых полей, порядок перечисления полей в списке имеет то же значение, что и в случае свойства IndexFieldNames;
- DescFields — список полей, сортируемых в порядке убывания, порядок перечисления полей имеет значение, указанное выше;
- CaseInsFields — список полей (строкового типа), которые следует сортировать независимо от регистра символов.

Формат перечисления полей в списках Fields, DescFields и CaseInsFields такой же, как и в случае свойства IndexFieldNames. Списки DescFields и CaseInsFields должны содержать поля из списка Fileds, но не обязательно совпадать с ним.

Кроме уже перечисленных свойств класса TIndexDef определенный интерес представляет также свойство Options. Данное свойство представляет собой набор (set) элементов. Некоторые из элементов этого свойства дублируют функции свойств, описанных выше. Элемент ixPrimary указывает, что данный индекс является первичным, а элемент ixUnique — что значения индекса должны быть уникальными. В листинге 15.3. приводится пример (на языке Delphi Language) создания индекса во время выполнения программы.

Листинг 15.3. Создание индекса

```
with ClientDataSet1.IndexDefs.AddIndexDef do
begin
  Name := 'Index1';
  Fields := 'CATEGORY;PRICE;NAME';
  DescFields := 'PRICE';
  CaseInsFields := 'NAME';
end;
```

Для того чтобы использовать один из созданных таким способом индексов, следует присвоить его имя свойству `IndexName`, например:

```
ClientDataSet1.IndexName := 'Index1';
```

Поиск в наборе данных

Поиск относится к числу наиболее часто используемых операций при работе с базами данных. К разновидности поиска можно отнести выбор записей по заданным критериям с помощью SQL-команды `select`. Компоненты, реализующие клиентские наборы данных, предоставляют ряд методов, позволяющих осуществлять расширенный поиск. Далее мы рассмотрим применение некоторых из этих методов.

Очевидно, самым простым методом поиска является поиск с помощью фильтров. Мы уже работали с фильтрами в предыдущей главе, здесь стоит упомянуть лишь некоторые дополнительные детали.

Для полей строкового типа фильтры позволяют использовать символы групповых операций (wildcards), регистро-независимые сравнения и функцию `SubString`, возвращающую фрагмент строки в указанной позиции. Возможности использования групповых символов в фильтрах довольно скромны и, по-видимому, основаны на той же функции `SubString`. Для того чтобы использовать регистро-независимые сравнения, следует присвоить значение `foCaseInsensitive` свойству `FilterOptions`.

Компоненты-наследники `TCustomClientDataSet` предоставляют еще несколько методов для поиска отдельных записей: `Locate`, `Lookup`, `FindKey` и `GotoKey`. Важно отметить, что поиск с помощью указанных методов может быть эффективным только в том случае, если поля, по которым осуществляется поиск, будут предварительно проиндексированы. Некоторые методы поиска записей вызовут исключительную ситуацию, если для заданных им полей отсутствуют индексы.

Метод `Locate` позволяет найти запись, содержащую определенные значения в определенных полях. У этого метода три параметра. Первый параметр — это строка, содержащая разделенный символами ";" список имен

полей записи, по значениям которых выполняется поиск. Второй параметр — переменная вариантного типа, содержащая список значений для указанных полей. Третий параметр позволяет указать параметры поиска, например, частичное совпадение и регистро-независимость (для строковых полей).

На языке Delphi Language вызов метода `Locate` может выглядеть так:

```
ClientDataSet1.Locate('CATEGORY;PRICE', VarArrayOf(['cpus', 120]),
[loCaseInsensitive]);
```

После этого вызова первая запись с указанными значениями указанных полей станет текущей. Напомним, что обратиться к полям текущей записи можно с помощью метода `FieldByName`.

Для языка C++ обращение к методу `Locate` выглядит сложнее (листинг 15.4).

Листинг 15.4 Вызов метода `Locate` в программе, написанной на языке C++

```
TLocateOptions Opts;
Opts.Clear();
Opts << loCaseInsensitive;
Variant values[2];
values[0] = Variant("cpus");
values[1] = Variant(120);
ClientDataSet1->Locate("CATEGORY;PRICE", VarArrayOf(values, 1), Opts);
```

Второй аргумент функции `VarArrayOf` в случае языка C++ указывает значение последнего индекса вариантного массива, переданного в качестве первого аргумента. Обычно это значение на единицу меньше числа элементов массива, так как индексация элементов начинается с нуля.

Метод `Locate` возвращает значение `true`, если ему удалось найти запись, соответствующую критериям поиска (и эта запись стала текущей), и `false` — в противном случае.

Метод `Lookup` позволяет получить значения полей записи, определенной значениями других полей. У этого метода также три параметра, первые два из которых соответствуют параметрам метода `Locate`. Третьим параметром метода `Lookup` является строка, содержащая разделенный символами ";" список имен полей, значения которых нужно получить. Метод `Lookup` возвращает переменную вариантного типа, содержащую значения полей, указанных в третьем параметре. В листинге 15.5 приводится пример вызова метода `Lookup` в программе, написанной на языке C++.

Листинг 15.5. Вызов метода `Lookup`

```
Variant values[2];
values[0] = Variant("CPUs");
values[1] = Variant(120);
Variant ResultFields = ClientDataSet1->Lookup("CATEGORY;PRICE",
VarArrayOf(values, 1), "ID;NAME;DISCOUNT");
```

В результате выполнения этого метода будет возвращен вариантный массив, содержащий значения полей `ID`, `NAME` и `DISCOUNT`.

Метод `FindKey` похож на метод `Locate`, но, в отличие от последнего, имеет только один параметр — вариантный массив, в котором передаются значения полей искомой записи в порядке, определенном структурой таблицы. Если соответствующая запись найдена, метод `FindKey` делает ее текущей и возвращает значение `true`. В противном случае, метод возвращает значение `false`.

Если мы не хотим использовать вариантные массивы для указания значений полей, мы можем воспользоваться другим механизмом, называемым *буфер поиска* (search key buffer). Для заполнения значений полей в буфере поиска используется тот же метод `FieldByName`, который служит для доступа к текущей записи. Для того чтобы указать, что мы вызываем метод `FieldByName` для заполнения буфера поиска, а не для модификации значений полей текущей записи, необходимо вызвать метод `SetKey`. Вызов этого метода переводит набор данных в особое состояние, обозначаемое `dsSetKey`. После того, как буфер поиска заполнен необходимыми значениями, для перехода к искомой записи используется метод `GotoKey`. Если запись, соответствующая критериям поиска, существует, она становится текущей. Пример поиска записи с помощью методов `SetKey` и `GotoKey` приводится в листинге 15.6.

Листинг 15.6. Пример использования метода `GotoKey`

```
ClientDataSet1->SetKey();
ClientDataSet1->FieldByName('CATEGORY')->Value = Variant('CPUs');
ClientDataSet1->FieldByName('PRICE')->Value = Variant(120);
ClientDataSet1->GotoKey();
```

Методам, рассмотренным здесь, присущи определенные ограничения (*см. ниже*), однако их достоинством является быстрота поиска. Данные методы способны выполнять быстрый поиск потому, что они используют индексы.

Закладки

Закладки (bookmarks) — особые элементы клиентского набора данных, позволяющие пометить определенную запись в наборе, а затем быстро перейти к ней. Для того чтобы создать закладку для текущей записи, следует вызвать метод `GetBookmark`. Метод `GetBookmark` возвращает значение типа `TBookmark`, идентифицирующее закладку. Для перехода к записи по закладке служит метод `GotoBookmark`, единственным параметром которого является идентификатор закладки. После вызова этого метода запись, для которой была определена переданная ему закладка, становится текущей. Удалить созданную закладку можно с помощью метода `FreeBookmark`.

Фильтрация данных с помощью события *OnFilterRecord*

Возможности поиска записей с помощью фильтров и других методов, предоставляемых компонентами, реализующими клиентские наборы данных, весьма ограничены. Возможность поиска при частичном совпадении строки ограничивается поиском полей, значения которых начинаются с заданной последовательности символов. Те же ограничения действуют и для групповых символов в фильтрах. Правда, возможности использования групповых символов в фильтрах можно несколько расширить при помощи события `OnFilterRecord`. Если значение свойства `Filtered` компонент-клиентского набора данных установлено равным `true`, это событие происходит при каждой активизации новой записи в наборе данных. Рассмотрим обработчик события `OnFilterRecord`, удаляющий записи с четным значением поля `ID` (листинг 15.7).

Листинг 15.7. Обработчик события `OnFilterRecord`

```
procedure TForm1.ClientDataSet1FilterRecord(DataSet: TDataSet;
var Accept: Boolean);
begin
  if Odd(DataSet.FieldByName('ID').AsInteger) then Accept := True
  else Accept := False;
end;
```

Параметр `DataSet` представляет фильтруемый набор данных, в котором фильтруемая запись является текущей. Если текущую запись следует оставить в наборе данных, параметру-переменной `Accept` присваивается значение `true`, в противном случае — `false`.

Неудобство, связанное с обработчиком события `OnFilterRecord`, заключается в том, что этот обработчик вызывается столько раз, сколько записей содержится в наборе данных (а число записей, как мы уже отмечали, может быть очень велико). Существует другой способ фильтрации данных, позволяющий выполнить эту операцию за один вызов процедуры.

Фильтрация данных с помощью компонента *DataSetProvider*

Если клиентский набор данных получает пакеты данных от компонента-провайдера, мы можем использовать события провайдера для фильтрации данных.

Событие компонента-провайдера `OnGetData` вызывается всякий раз, когда провайдер готов отправить пакет данных клиенту. Обработчику этого события передается указатель на объект класса `TCustomClientDataSet`, содержащий записи пакета данных. Обработчик может выполнять *любые* манипуляции с записями, и именно модифицированный пакет данных будет передан клиенту.

Напишем на языке Delphi Language приложение, осуществляющее поиск в базе данных с помощью события `OnGetData` компонента-провайдера. Создадим приложение просмотра базы данных на основе компонентов `SQLConnection`, `SQLDataSet`, `DataSetProvider`, `ClientDataSet` и `DBGrid`. Отдельные этапы создания подобного приложения описывались так часто, что останавливаться на них мы не будем. Рассмотрим только те элементы, которые следует добавить для обеспечения функции поиска. Создаваемое приложение позволяет осуществлять поиск записей по частично заданным значениям отдельных полей, при этом будут найдены все записи, указанные поля которых включают заданную последовательность символов в *любом* месте. Для выбора поля, по значениям которого производится поиск, используется раскрывающийся список (объект `FieldComboBox`). Последовательность символов, которую должно содержать указанное поле в искомых записях, вводится с помощью компонента `Edit` (объект `ValueEdit`). После нажатия кнопки **Найти** (объект `FindButton`) в таблице `DBGrid` отображаются записи, соответствующие критериям поиска. Исходный текст приложения приводится в листинге 15.8.

Листинг 15.8. Приложение, фильтрующее данные

```
unit Unit1;

interface

uses
  SysUtils, Types, Classes, Variants, QTypes, QGraphics, QControls,
  QForms, QDialogs, QStdCtrls, dbExpress, Provider, SqlExpr, QExtCtrls,
  QGrids, QDBGrids, DB, DBClient, DBLocal, DBLocalS, FMTBcd;

type
```

```
  TForm1 = class(TForm)
    SQLConnection1: TSQLConnection;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    Panel1: TPanel;
    FieldComboBox: TComboBox;
    ValueEdit: TEdit;
    FindButton: TButton;
    ClientDataSet1: TClientDataSet;
    DataSetProvider1: TDataSetProvider;
    SQLDataSet1: TSQLDataSet;
    procedure FormShow(Sender: TObject);
    procedure FindButtonClick(Sender: TObject);
    procedure DataSetProvider1GetData(Sender: TObject;
      DataSet: TCustomClientDataSet);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure FillCombo;
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.FillCombo;
var
  i : Integer;
begin
  for i := 0 to ClientDataSet1.FieldDefs.Count-1 do
  FieldComboBox.Items.Add(ClientDataSet1.FieldDefs.Items[i].Name);
end;

procedure TForm1.FormShow(Sender: TObject);
```

```
begin
  FillCombo;
end;

procedure TForm1.FindButtonClick(Sender: TObject);
begin
  ClientDataSet1.Refresh;
end;

procedure TForm1.DataSetProvider1GetData(Sender: TObject;
  DataSet: TCustomClientDataSet);
var
  S : String;
begin
  if FieldComboBox.Text = '' then Exit;
  DataSet.First;
  while not DataSet.Eof do
  begin
    S := DataSet.FieldByName(FieldComboBox.Text).AsString;
    if Pos(ValueEdit.Text, S) = 0 then DataSet.Delete
    else DataSet.Next;
  end;
end;
end.
```

Процедура `FillCombo` заполняет список `FieldComboBox` названиями столбцов таблицы. Нажатие кнопки `FindButton` вызывает метод `Refresh` объекта `ClientDataSet1`, в результате чего клиентский набор данных посылает провайдеру запрос на выдачу нового пакета данных. Перед тем, как передать пакет данных клиенту, провайдер вызывает обработчик `DataSetProvider1GetData`. В этом обработчике мы последовательно (с помощью метода `Next`) проверяем все записи на соответствие критерию поиска и *удаляем* те записи, которые этому критерию не соответствуют (с помощью метода `Delete`, удаляющего текущую запись). Таким образом, клиенту будет передан набор данных, содержащий только искомые записи. Возникает вопрос: может ли удаление записей из набора данных повлиять каким-то образом на содержимое базы данных? Нет, не может. Ведь выполняемые операции не отображаются в пакете `Delta` (собственно, в этот момент пакет `Delta` еще и не существует). Если затем пользователь внесет изменения в записи, будет создан пакет `Delta`, отражающий только те изменения, которые были внесены пользователем явным образом.

Работающее приложение показано на рис. 15.2.

Рис. 15.2. Приложение фильтрации данных отображает только те записи, в поле `NAME` которых содержится слово "Mouse"

Обработчик `OnGetData` может использоваться не только для фильтрации данных, но и для любой их предварительной обработки, включая перекодировку значений строковых полей из одной кодовой таблицы в другую.

После вызова метода клиентского набора данных `ApplyUpdates` компонент-провайдер отправляет полученный от клиента пакет данных источнику данных. При этом вызывается событие `OnUpdateData`, обработчик которого имеет формат, аналогичный формату обработчика события `OnGetData`. Обработчик `OnUpdateData` позволяет внести изменения в данные перед тем, как они будут отправлены базе данных.

Редактирование записей и метод *Post*

Если текущая запись клиентского набора данных редактируется при помощи метода `FieldByName`, то для "закрепления" внесенных изменений следует вызвать метод `Post`. Такие методы, как `Edit`, `InsertRecord` или `Append`, вызывают метод `Post` автоматически. Необходимость в явном вызове метода `Post` может возникнуть в ситуации, представленной в листинге 15.9.

Листинг 15.9. Добавление новой записи в набор данных

```
void __fastcall TForm1::Button1Click(TObject * Sender);
{
  // Создаем новую пустую запись, которая становится текущей
  ClientDataSet1->Insert();
  // Заполняем поля записи
  ClientDataSet1->FieldByName("ID")->Value =  Variant(30);
```

```
  ClientDataSet1->FieldByName("CATEGORY")->Value =  Variant("CPUs");
  ...
  ClientDataSet1->Post();
}
```

Следует помнить, что метод `Post` не обновляет содержимое первичного источника данных, так что для "окончательного" внесения изменений следует вызвать метод `ApplyUpdates` или `SaveToFile`, в зависимости от используемой модели приложения.

Компоненты графического интерфейса приложений баз данных

Компоненты, предназначенные для создания графического интерфейса приложений баз данных, отличаются от обычных компонентов тем, что могут связываться с источником данных (компонентом `DataSource`) с помощью имеющегося у всех этих компонентов свойства `DataSource` и динамически отображать изменения в соответствующем наборе данных. Все компоненты, рассмотренные в этом разделе, расположены на странице **Data Controls** палитры компонентов.

Компонент *DBGrid*

Компонент `DBGrid`, с которым мы уже сталкивались неоднократно, является наиболее часто используемым компонентом графического интерфейса приложений баз данных. Как мы уже знаем, компонент `DBGir` представляет собой таблицу, позволяющую отображать и редактировать содержимое таблицы базы данных. И, конечно, мы уже знакомы с поведением этого компонента: активная запись помечается стрелкой, запись, редактируемая в данный момент — символом, напоминающим текстовый курсор, а новая запись, еще не внесенная в клиентский набор данных, — астериском. Многие свойства и методы компонента `DBGrid` являются аналогами свойств компонента `StringGrid`, и останавливаться на них мы не будем. Отметим лишь те свойства, которые являются специфичными для компонента, работающего с базами данных.

Свойство-массив `Fields` содержит список всех полей (столбцов) таблицы. Элементами массива являются объекты класса `TField`. Нумерация полей начинается с нуля. Общее число полей таблицы можно узнать с помощью свойства `FieldCount`.

Свойство `ReadOnly` управляет возможностью изменять содержимое набора данных с помощью компонента `DBGrid`. Если в нашем приложении компонент `DBGrid` предназначен только для просмотра данных, назначим этому свойству значение `false`.

Свойство Columns содержит список описаний столбцов таблицы. Элементами этого списка являются объекты класса TColumn. Среди свойств этого класса особого внимания заслуживают следующие:

- Field — указатель на объект TField, соответствующий полю набора данных, отображаемому в данном столбце таблицы;
- FieldName — это свойство позволяет изменить поле набора записей, сопоставленное данному столбцу (по умолчанию столбцы таблицы сопоставляются полям набора данных в том порядке, в котором эти поля определены в структуре набора данных);
- Title — это свойство-объект класса TColumnTitle, которое позволяет настроить внешний вид заголовочной ячейки столбца: выбрать для нее название (по умолчанию столбец называется также, как и представляемое им поле набора данных), установить цвет фона и параметры шрифта;
- Font — это свойство позволяет выбрать шрифт для ячеек столбца, содержащих данные;
- Color — позволяет выбрать цвет фона для ячеек столбца, содержащих данные;
- ReadOnly — это свойство позволяет запретить пользователю изменение данных в соответствующем столбце.

Свойство SelectedField позволяет переместить фокус ввода на определенное поле (для этого свойству назначается указатель на соответствующий объект TField), то же самое можно сделать с помощью свойства SelectedIndex, которому следует указать порядковый номер поля. Эти свойства также позволяют определить, какой ячейке принадлежит фокус ввода в данный момент.

С помощью свойства Columns можно настроить внешний вид отдельных столбцов таблицы. А как настроить вид отдельных строк или ячеек в зависимости от их содержимого? Для этого необходимо воспользоваться обработчиком события OnDrawColumnCell. Это событие вызывается для отрисовки каждой ячейки таблицы.

Рассмотрим пример. Допустим, мы хотим, что записи таблицы, в которых значения поля PRICE превышают 250, выделялись красным цветом. Решим эту задачу с помощью обработчика события OnDrawColumnCell (см. листинг 15.10). Для того, чтобы использовать этот обработчик, необходимо присвоить значение false свойству DefaultDrawing объекта TDBGrid. В этом случае весь процесс отрисовки содержимого ячеек таблицы будет возложен на программиста. Однако нам не придется самостоятельно выполнять все операции по выводу значений. Как мы увидим из листинга, большую часть работы сделает за нас метод DefaultDrawColumnCell, нам же придется изменить лишь некоторые свойства объекта Canvas нашего компонента DBGrid.

Листинг 15.10. Обработчик события `OnDrawColumnCell`

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
const Rect: TRect; DataCol: Integer; Column: TColumn;
State: TGridDrawState);
var
  DBGrid : TDBGrid;
  DataSet : TCustomClientDataSet;
begin
  DBGrid := Sender as TDBGrid;
  DataSet := DBGrid.DataSource.DataSet as TCustomClientDataSet;
  if DataSet.FieldByName('PRICE').AsInteger > 250 then
  DBGrid.Canvas.Font.Color := clRed
  else
  DBGrid.Canvas.Font.Color := clBlack;
  DBGrid.DefaultDrawColumnCell(Rect, DataCol, Column, State);
end;
```

Самое интересное в этом листингето, как мы получаем доступ к значениям ячеек таблицы. У компонента `DBGrid` нет свойства `Cells`, которое содержало бы эти значения, и получить их можно только из набора данных, с которым компонент `DBGrid` связан через компонент `DataSourse`. Параметр `Sender` нашего обработчика содержит ссылку на экземпляр `TDBGrid`, вызывавший обработчик. Мы инициализируем значением параметра `Sender` переменную `DBGrid`, а затем получаем указатель на объект-набор данных. Как же получить содержимое ячейки, для которой вызван обработчик? Строке таблицы, в которой находится ячейка, соответствует активная запись в наборе данных (доступ к полям этой записи можно получить с помощью метода `FieldByName`). Что же касается поля (столбца) таблицы, содержащего ячейку, то его индекс передается обработчику в параметре `DataCol`, а параметр `Column` содержит ссылку на объект, описывающий столбец таблицы. Так как в нашем обработчике мы изменяем вид всех ячеек строки, нам не нужно знать, для какого столбца вызван обработчик.

Мы проверяем значение поля `PRICE` для текущей строки и на основании этого значения устанавливаем значение свойства `Color` объекта `Font` объекта `Canvas`. После этого нам остается только вызвать метод `DefaultDrawColumnCell`, передав ему параметры, переданные обработчику события.

Компонент *DBNavigator*

Этот часто используемый компонент служит как для перемещения по набору данных, так и для создания, ввода или удаления записей. Компонент `DBNavigator` представляет собой панель, содержащую набор кнопок, связанных с наиболее часто используемыми командами при работе с наборами данных. Для того чтобы использовать компонент `DBNavigator`, достаточно указать в свойстве `DataSource` соответствующего объекта объект-источник данных. Если с указанным объектом-источником связаны и другие компоненты пользовательского интерфейса, то их содержимое (значение полей текущей записи) будет меняться в зависимости от команд, посылаемых при помощи кнопок навигатора. Внешний вид панели навигатора показан на рис. 15.3.

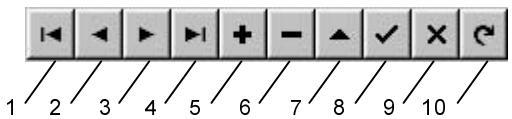


Рис. 15.3. Панель компонента `DBNavigator`

Кнопки панели навигатора позволяют выполнять следующие команды (номера соответствуют кнопкам, приведенным на рис. 15.3):

- кнопка 1 — переход к первой записи в наборе;
- кнопка 2 — переход к предыдущей записи (относительно текущей);
- кнопка 3 — переход к следующей записи (относительно текущей);
- кнопка 4 — переход к последней записи в наборе;
- кнопка 5 — вставка новой записи в месте расположения указателя;
- кнопка 6 — удаление записи в месте расположения указателя;
- кнопка 7 — редактирование текущей записи;
- кнопка 8 — внесение записи в клиентский набор данных;
- кнопка 9 — отмена редактирования (до того, как запись была внесена в клиентский набор);
- кнопка 10 — обновление набора данных.

Относительно кнопки 8 необходимо сделать некоторые пояснения. Эта кнопка вносит запись в набор данных, но не вызывает методов `ApplyUpdates` или `SaveToFile`. Поэтому, если для внесения изменений в первичный источник данных необходимо вызвать метод `ApplyUpdates`, как это имеет место в случае с базами данных, связанными через компоненты dbExpress, или метод `SaveToFile`, как в случае использования локального файла в режиме явного сохранения, внесенная в набор запись не будет внесена в первичное хранилище данных до тех пор, пока не будет вызван соответствующий ме-

тод. В случае, если используется схема MyBase с автоматическим сохранением изменений, нажатие на эту кнопку приведет к немедленному изменению в хранилище данных.

На первый взгляд, взаимодействие между компонентом DBNavigator и, например, компонентом DBGrid может показаться "магическим". На самом деле никакой магии здесь нет. Компонент DBNavigator вносит изменения в набор данных, с которым он связан посредством компонента DataSource, а компонент DBGrid, связанный с тем же набором данных, динамически отображает внесенные изменения. Все происходило бы точно также, как если бы изменения вносились в набор данных не с помощью компонента DBNavigator, а непосредственно при помощи методов клиентского набора данных Edit, Insert, Post или Delete.

Среди свойств компонента DBNavigator следует отметить булево свойство ConfirmDelete, которое позволяет установить режим вывода диалогового окна с запросом на подтверждение удаления записи из набора данных и свойство Hints, представляющее собой список "всплывающих подсказок" для кнопок панели DBNavigator. С помощью этого свойства мы можем русифицировать "всплывающие подсказки".

Компонент *DBText*

Этот компонент является аналогом компонента Label, т. е. позволяет отображать значение отдельного поля набора данных без возможности редактирования. Для того чтобы использовать этот компонент, необходимо назначить свойству DataSource ссылку на компонент-источник данных, а свойству DataField название поля, значения которого должен отображать компонент. После этого компонент DBText будет отображать значение указанного поля для активной записи набора данных. Получить доступ к текущему значению, отображаемому компонентом BDText, можно с помощью свойства Field.

Компонент *DBEdit*

Данный компонент позволяет редактировать значение отдельного поля текущей записи. Собственные свойства компонента DBEdit практически не отличаются от свойств компонента DBText. Дополнительное свойство Readonly позволяет запретить редактирование данных.

Компонент DBEdit является наследником класса TCustomMaskEdit и позволяет проверять соответствие введенной информации заданной маске с помощью метода ValidateEdit, который в случае несоответствия маски и текущих данных генерирует исключительную ситуацию. Маска определяется свойством EditMask поля записи, связанного с компонентом (но нескрытым свойством EditMask класса TCustomMaskEdit), а значение, созданное на основе маски, хранится в свойстве EditText.

Компонент *DBMemo*

С помощью этого компонента отображается содержимое двоичного поля записи (например, BLOB), если это поле содержит многострочный текст. Данный компонент позволяет не только отображать содержимое поля, но и редактировать его. Кроме уже известных нам свойств `DataField`, `DataSource`, `Field` и `ReadOnly`, компонент `DBMemo` включает свойство `AutoDisplay`. Если мы хотим, чтобы содержимое соответствующего поля загружалось в компонент автоматически, назначим этому свойству значение `true`.

Необходимость в свойстве `AutoDisplay` вызвана тем, что двоичные поля, такие как BLOB, могут содержать данные различных типов, так что, если тип столбца таблицы определен как BLOB, в этом столбце может содержаться и текст и графика, и другие данные. Можно использовать режим `AutoDisplay`, равным `true`, если мы уверены, что все ячейки столбца содержат только текстовые данные. Если свойству `AutoDisplay` присвоено значение `false`, для загрузки данных следует использовать метод `LoadMemo`.

Компонент *DBImage*

Как и компонент `DBMemo`, этот компонент предназначен для считывания данных из двоичных полей, но служит для отображения графики, а не текста. Подробно описывать компонент `DBImage` мы здесь не будем, так как ниже приводится пример приложения, использующего этот компонент.

Компонент *DBListBox*

Этот компонент применяется для полей, допускающих небольшое множество заранее известных значений (как поле `CATEGORY` в рассмотренных выше примерах). Список допустимых значений готовится заранее и заносится в свойство `Items` типа `TStrings`.

Компонент *DBComboBox*

Как и компонент `DBList`, компонент `DBComboBox` позволяет отображать текущее значение и выбирать новое значение поля в раскрывающемся списке. Набирая текст в области ввода, можно быстро переместиться к нужной записи в списке или ввести значение, отсутствующее в нем.

В дополнение к свойствам, аналогичным свойствам компонента `ComboBox`, в классе `TDBComboBox` определено свойство `Style` типа `TComboBoxStyle`, позволяющее определить способ отображения элементов в списке.

Напишем приложение, позволяющее вносить изменения в таблицу `idpricelist`, описанную в главе 14. Для удобства ввода новых записей мы

введем в приложение функцию автоматического заполнения поля ID для новой записи и возможность выбора значения поля CATEGORY из списка DBComboBox. Создадим в новом проекте на языке C++ цепочку из компонентов SQLConnection, SQLClientDataSet, DataSource, DBGrid и DBNavigator. Связи между объектами приложения показаны на диаграмме рис. 15.4.

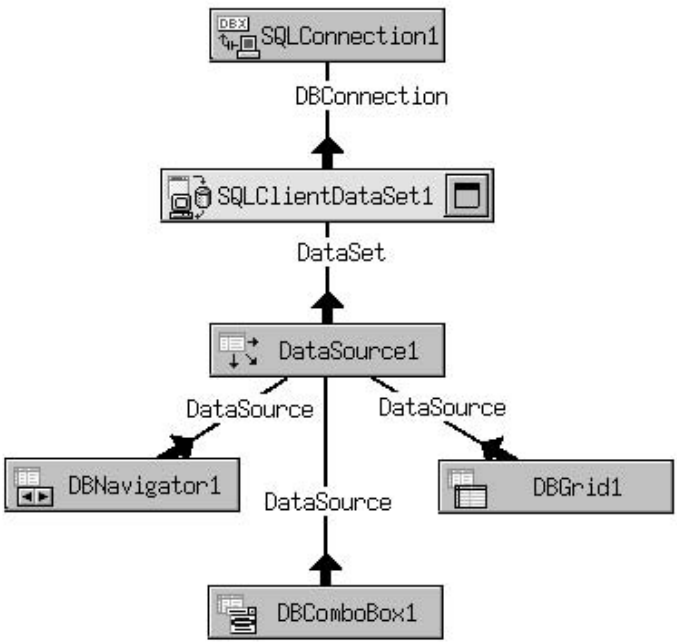


Рис. 15.4. Связи между компонентами приложения редактирования таблицы idpricelist

Настроим объект SQLConnection1 для связи с базой данных, содержащей таблицу idpricelist, установим свойство Connected, равным true. Назначим свойству ComandText объекта SQLClientDataSet1 команду:

```
select * from idpricelist
```

и сделаем свойство Active этого объекта, равным true.

Список DBComboBox1 можно разместить где угодно, но мы сделаем так, чтобы этот список заменял собой редактируемую ячейку столбца CATEGORY. В инспекторе объектов назначим свойству Style объекта DBComboBox1 значение csDropDownList. Свойству Visible присвоим значение false, так как список будет отображаться только во время редактирования ячейки, а свойству DataField — значение CATEGORY. Откроем редактор свойства Items объекта DBComboBox1 и введем в нем строки, соответствующие возможным значениям поля CATEGORY.

Для того, чтобы реализовать указанную схему работы со списком, нам опять придется взять на себя отрисовку ячеек таблицы. Присвоим значение `false` свойству `DefaultDrawing` объекта `DBGrid1`. Исходный текст методов класса `TForm1` создаваемого приложения приводится в листинге 15.11.

Листинг 15.11. Методы класса `TForm1`

```
//---------------------------------------------------------------------
#include <clx.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"
TForm1 *Form1;
int MaxId;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  SQLClientDataSet1->Last();
  MaxId = SQLClientDataSet1->FieldByName("ID")->AsInteger;
}
//---------------------------------------------------------------------

void __fastcall TForm1::DBGrid1DrawColumnCell(TObject *Sender,
      const TRect &Rect, int DataCol, TColumn *Column,
      TGridDrawState State)
{
  if (State.Contains(gdFocused))
  {
    if ((DataCol == 1) && (SQLClientDataSet1->State == dsEdit))
    {
      TDataSet * DataSet = DBGrid1->DataSource->DataSet;
      DBComboBox1->Text = (String)DataSet->FieldByName("CATEGORY")↵
      ->AsString;
      DBComboBox1->Height = Rect.Height();
      DBComboBox1->Width = Rect.Width();
```

```
      TPoint Point;
      Point.x = Rect.left;
      Point.y = Rect.top;
      Point = DBGrid1->ClientToParent(Point, this);
      DBComboBox1->Left = Point.x;
      DBComboBox1->Top = Point.y;
      DBComboBox1->Show();
    }
    else
    {
      DBComboBox1->Hide();
      DBGrid1->DefaultDrawColumnCell(Rect, DataCol, Column, State);
    }
  }
  else
  {
    DBGrid1->DefaultDrawColumnCell(Rect, DataCol, Column, State);
  }
}

//---------------------------------------------------------------------------

void __fastcall TForm1::UpdateButtonClick(TObject *Sender)
{
  SQLClientDataSet1->ApplyUpdates(-1);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::SQLClientDataSet1AfterInsert(TDataSet *DataSet)
{
  DataSet->FieldByName("ID")->Value = Variant(++MaxId);
}
//---------------------------------------------------------------------------
```

Переменная `MaxId` понадобится нам для автоматического заполнения поля `ID`. Она содержит текущее максимальное значение данного поля. Переменная `MaxId` сделана глобальной для наглядности, чтобы не вносить ее в заголовочный файл. Начальное значение этой переменной присваивается в конструкторе класса. Присвоение значения полю `ID` новой записи выполняется в обработчике события `OnAfterInsert` (`SQLClientDataSet1AfterInsert`), а управление

элементом DBComboBox1 — в обработчике DBGrid1DrawColumnCell. В этом обработчике мы сначала проверяем, имеет ли текущая ячейка фокус ввода. Если нет, мы вызываем стандартный метод отрисовки ячейки, независимо от того, что это за ячейка. Если текущая ячейка владеет фокусом ввода, мы проверяем, принадлежит ли она столбцу CATEGORY (DataCol == 1) и находится ли набор данных в состоянии редактирования. Если оба эти условия выполнены, мы выводим список DBComboBox1 на месте ячейки с размерами, соответствующими размерам ячейки. В противном случае мы вызываем стандартный метод отрисовки. Таким образом, раскрывающийся список в поле ячейки выводится только в том случае, если таблица находится в состоянии редактирования (для того, чтобы перевести таблицу в это состояние, следует нажать кнопку **Редактировать** компонента DBNavigator). Работающее приложение показано на рис. 15.5.

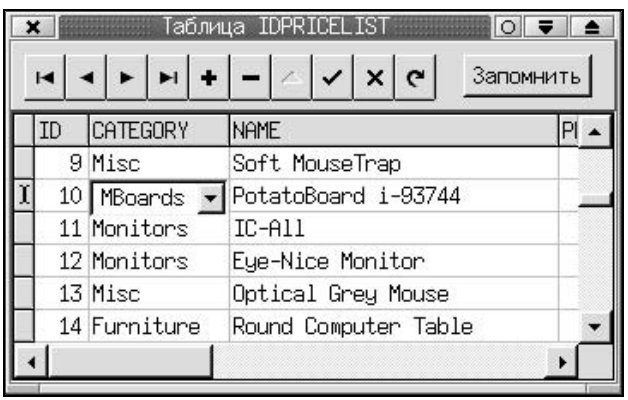


Рис. 15.5. Работающее приложение с раскрывающимся списком в поле ячейки

Кнопка **Запомнить** (объект UpdateButton) служит для вызова метода ApplyUpdates, вносящего изменения в базу данных.

Компонент *DBCheckBox*

С помощью этого компонента можно отображать и изменять значение поля таблицы, имеющего тип Boolean. Этот компонент можно применять также и для редактирования любых полей, которые могут принимать одно из двух возможных произвольных значений. Список значений, при которых компонент будет считаться "включенным", задается в свойстве ValueChecked. Значения этого списка должны быть разделены символом ";", например:

```
DBCheckBox1->ValueChecked = "True;Yes";
```

Таким же образом определяется и список значений, при которых компонент будет считаться "выключенным" (этот список следует назначить свой-

ству ValueUnchecked). Регистр символов значений в этих списках не учитывается.

Если поле таблицы имеет логический тип, строки, занесенные в свойства ValueChecked и ValueUnchecked, игнорируются, а состояние компонента определяется только значением поля таблицы: значение true - включено, значение false — выключено.

Компонент *DBRadioGroup*

Этот компонент позволяет отображать и редактировать значения полей, допускающих ограниченный набор возможных значений, в виде набора переключателей. Каждому значению можно поставить в соответствие один из переключателей группы. Список названий переключателей заносится в свойство Items, а список соответствующих им значений — в свойство Values (оба свойства имеют тип TStrings). Значение поля, связанного с компонентом, можно получить при помощи свойства Value.

Хранение изображений в базах данных

В этом разделе мы напишем приложение, позволяющее заносить изображения в базу данных, и компонент DBImage, позволяющий просматривать такие изображения.

Прежде всего нам необходимо создать таблицу, способную хранить изображения. Лучше всего для хранения изображений подходят поля типа BLOB. BLOB представляет собой большой (очень большой) двоичный объект. Фактически в полях типа BLOB можно хранить любые данные, нужно только позаботиться о корректных средствах их отображения и редактирования.

Создадим новую базу данных или откроем уже имеющуюся, и в программе-клиенте SQL дадим команду:

```
create table images(name varchar(32) not null unique, image blob);
```

Мы создали таблицу images с двумя столбцами: name типа "строка переменной длины" и image типа BLOB. Почему поле name должно быть непустым и не должно содержать повторяющихся значений, мы объясним позднее. Сейчас отметим только, что для удобства работы с таблицами, содержащими BLOB, с помощью компонентов Kylix запись таблицы должна содержать хотя бы одно поле, не допускающее повторяющихся значений.

Создадим приложение просмотра и редактирования баз данных, содержащих изображения, на основе компонента SQLClientDataSet. Компонент DBGrid не позволяет отображать содержимое полей BLOB, поэтому использовать его в нашем приложении мы не будем. Вместо этого воспользуемся компонентом DBEdit для отображения и редактирования содержимого поля

name и компонентом DBImage для отображения содержимого поля image. Схема связей между объектами приложения показана на рис. 15.6. Свойствам DataField объектов DBEdit1 и DBImage1 следует присвоить значения NAME и IMAGE соответственно.

Вставка изображений в поля таблицы представляет определенные сложности. В принципе, компонент DBImage предназначен не только для просмотра, но и для редактирования изображений, хранящихся в BLOB-полях баз данных. Для этого предусмотрены методы CopyToClipboard, CutToClipboard и PasteFromClipboard, однако на практике не все изображения можно вставить в BLOB-поля при помощи этих методов, и, вообще, можно сказать, что компонент DBImage больше подходит для просмотра изображений, нежели для их модификации.

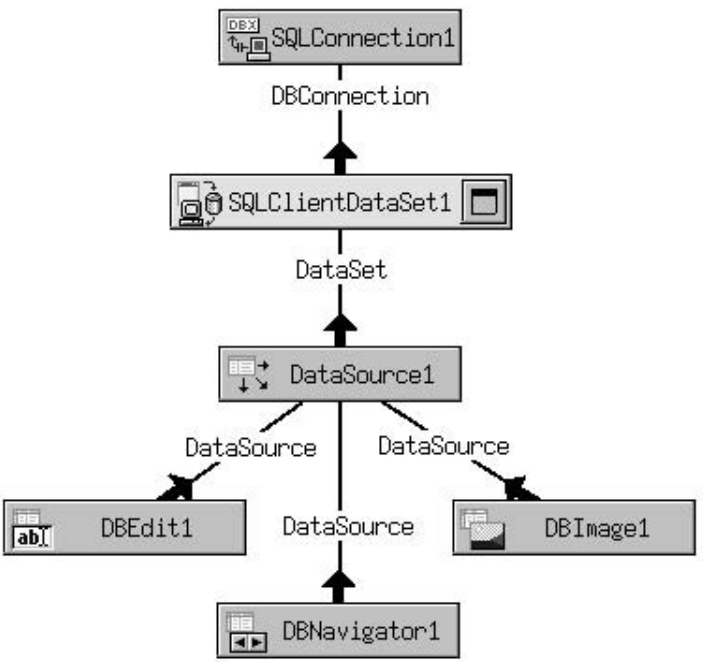


Рис. 15.6. Связи между объектами в приложении просмотра изображений

Решить проблему заполнения BLOB-полей таблицы изображениями (как и другими данными) можно достаточно просто с помощью методов класса TBlobField. Этот класс является потомком класса TField и специально предназначен для работы с BLOB-полями. Добавим в наше приложение кнопку **Открыть** (объект OpenButton класса TButton), позволяющую загрузить в BLOB-поле текущей записи таблицы изображение из графического файла, хранящегося на диске.

Обработчик события OnClick этой кнопки приводится в листинге 15.12.

Листинг 15.12. Обработчик события OnClick кнопки OpenButton

```
void __fastcall TForm1::OpenButtonClick(TObject *Sender)
{
  if (OpenDialog1->Execute())
  {
    SQLClientDataSet1->Edit();
    TBlobField * BLOB =
      (TBlobField *)SQLClientDataSet1->FieldByName("IMAGE");
    BLOB->LoadFromFile((String)OpenDialog1->FileName);
  }
}
```

В обработчике мы вызываем метод Edit клиентского набора данных, но не вызываем метод Post. Связано это с тем, что при заполнении поля image данными только что вставленной записи, поле name может быть незаполненным, а, в соответствии с определением таблицы images, это поле не должно быть пустым.

Как видите, для загрузки данных в BLOB-поле мы используем метод LoadFromFile класса TBlobField (имя файла выбирается с помощью диалогового окна компонента OpenDialog). Метод LoadFromFile может загружать данные из любых файлов, поэтому на этапе загрузки у нас не возникнет проблем с форматом файла, главное, чтобы данный формат поддерживался отображающим компонентом DBImage. Компонент DBImage способен отображать файлы всех форматов, поддерживаемых лежащей в его основе системой Qt library (BMP, JPG, PNG и др.).

Для того, чтобы присвоить BLOB-полю содержимое объекта класса Tpicture, можно воспользоваться методом LoadFromStream класса TBlobField, как это показано в листинге 15.13.

Листинг 15.13. Передача данных объекта TPicture объекту TBlobField

```
TBlobField * BLOB =
  (TBlobField *)SQLClientDataSet1->FieldByName("IMAGE");
TMemoryStream * MS = new TMemoryStream();
Picture->Bitmap->SaveToSream(MS);
BLOB->LoadFromStream(MS);
delete MS;
```

Методы класса TBlobField SaveToFile и SaveToStream позволяют считывать данные BLOB-поля.

Выше отмечалось, что записи таблиц, содержащих BLOB-поля, должны иметь поля, не допускающие одинаковых значений. Связано это с тем, что в таких операциях с записями, как удаление, поля идентифицируются по значениям их записей. При этом компоненты, работающие с наборами записей, не сравнивают содержимое BLOB-полей, так что записи, различающиеся только содержимым полей типа BLOB, будут "казаться" программе одинаковыми. Такая ситуация приведет к выводу сообщений об ошибках при попытке выполнения операций редактирования или удаления записей.

Приложения баз данных и XML-документы

В этом разделе речь пойдет о специфической возможности, присущей программам, создаваемым с помощью средств разработки от Borland — переносе данных между XML-документами и базами данных. Для решения этой задачи Borland Kylix предоставляет утилиту XML Mapper и несколько специальных компонентов, которые и будут рассмотрены ниже.

Утилита XML Mapper и компонент *XMLTransformProvider*

Выше уже отмечалось, что компоненты для работы с базами данных могут использовать XML-документы в качестве источника данных. Для преобразования XML-документов произвольной структуры в форматы данных, с которыми могут работать компоненты БД, служит утилита XML Mapper.

Утилита XML Mapper позволяет выполнять следующие преобразования:

- XML-документа в пакет данных;
- XML-документа в "дельта-пакет", позволяющий добавлять записи в базу данных на основе элементов XML-документов;
- XML-документа в "дельта-пакет", позволяющий удалять записи из базы данных на основе элементов XML-документов;
- XML-документа в пакет данных, содержащий набор значений элементов ввода;
- пакета данных в XML-документ.

Утилита XML Mapper позволяет сохранять результаты преобразований как в форме XML-документов и файлов, содержащих пакеты данных, так и в форме *файлов преобразования* (по умолчанию эти файлы имеют расширение .xtr). Файлы преобразования не содержат данных. Вместо этого они содержат инструкции, с помощью которых компоненты Kylix могут выполнять

соответствующие преобразования во время работы приложений, используя данные, с которыми работает приложение. В этом смысле файлы преобразования можно сравнить с XSL-шаблонами, содержащими инструкции, с помощью которых компонент `XSLPageProducer` преобразует XML-документы в HTML-файлы.

Для начала напишем простое приложение, позволяющее работать с данными, хранящимися в XML-документе, с помощью компонентов `ClientDataSet` и `DBGrid`. Компонент `ClientDataSet` нуждается в компоненте-провайдере, в качестве которого в данном случае мы используем компонент `XMLTransformProvider`. Кроме XML-файла, содержащего данные, компоненту `XMLTransformProvider` необходим файл преобразований, с помощью которого этот компонент сможет преобразовать набор элементов XML-данных в пакет данных, который затем будет передан клиентскому набору данных. Необходимый файл преобразований мы создадим с помощью утилиты XML Mapper. В качестве документа-источника XML-данных мы используем файл catalog.xml (см. листинг 14.5). Заполним этот файл данными соответствующего формата (если мы не сделали этого раньше).

Запустим утилиту XML Mapper. Как видим, окно этой графической утилиты (рис. 15.7) разделено на три части. Левая часть (окно **Document**) служит для отображения исходного XML-документа. Центральная часть (окно **Transformation**) позволяет задавать параметры, необходимые для генерации пакетов данных и файлов преобразования. Правая часть окна XML Mapper (окно **Datapacket**) позволяет просматривать результирующие пакеты данных.

С помощью кнопки **Open** откроем файл catalog.xml. Его структура отображается в окне **Document**. У окна **Document** две вкладки: **Schema View** и **Document View**. Мы воспользуемся последней из этих вкладок.

На вкладке **Document View** в форме древовидной структуры представлены элементы данных XML-документа catalog.xml. Раскроем группы `Catalog` и `Item`, перейдем к центральному окну утилиты и выберем вкладку **Mapping**. В окне **Document** щелкнем кнопкой мыши по элементам `id`, `category`, `name`, `price` и `discount`. Соответствующие элементы будут появляться в списке окна **Transformation** (поле **Selected Nodes**).

Теперь в центральном окне перейдем на вкладку **Node Properties**. Эта вкладка позволяет указывать свойства полей будущего набора данных. Поле, для которого назначаются параметры, следует выбирать в окне **Document**. Выберем поле `id`. Свойству `DataType` этого поля назначим значение `Integer`. Назначим значения `true` свойствам `Value Required` и `In Primary Key`.

Свойства остальных полей, которые необходимо отредактировать, приводятся в табл. 15.1.

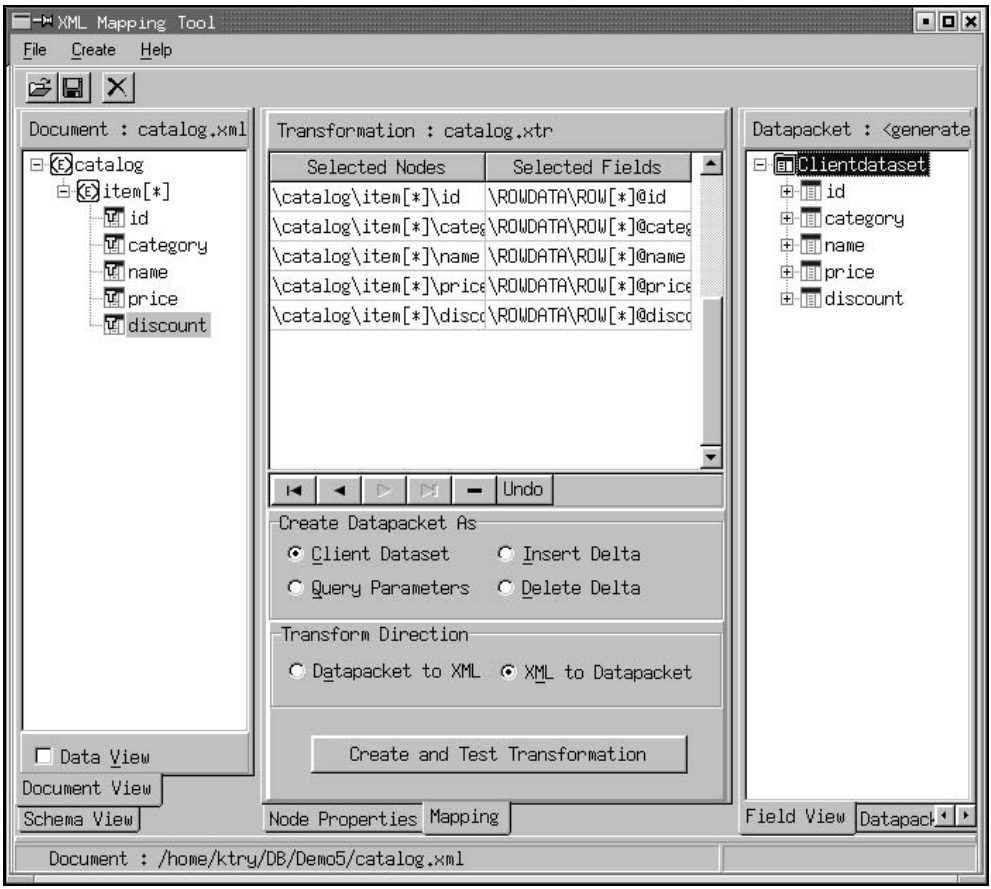


Рис. 15.7. Утилита XML Mapper

Таблица 15.1. *Свойства полей записей для преобразования файла catalog.xml*

Свойство	Значение
Поле `category`	
`Data Type`	`String`
`Max Length`	10
Поле `name`	
`Data Type`	`String`
`Max Length`	32

Таблица 15.1 *(окончание)*

Свойство	Значение
Поле `price`	
`Data Type`	`Integer`
`Value Required`	`True`
Поле `discount`	
`Data Type`	`Integer`

Теперь вернемся на вкладку **Mapping** окна **Transformation** и в группе **Create Datapacket as** выберем пункт **Create Dataset** (создание набора данных). В группе **Transform Direction** выберем пункт **XML to Datapacket**. Выберем команду меню **Create | Datapacket from XML**. После этого в окне **Datapacket** на вкладке **Field View** появится древовидная схема, отображающая структуру определенного нами пакета данных. Теперь можем нажать на кнопку **Create and Test Transformation**. При этом открывается окно с компонентом `DBGrid`, содержащим элементы созданного набора данных. Если мы видим, что полученный набор данных не соответствует нашим требованиям, мы можем вернуться к любому из этапов создания преобразования и изменить его.

После того как работа с набором данных закончена, выберем команду меню **File | Save | Transformation**. В открывшемся окне выберем имя файла преобразования (xtr-файл).

Мы создали файл преобразования, позволяющий конвертировать данные из произвольного XML-формата в пакет данных. С помощью этого преобразования мы сможем просматривать содержимое XML-документа, например, с помощью компонента DBGrid. Но мы хотим не только просматривать содержимое XML-файла, но и вносить в него изменения. Для этого нам понадобится еще один файл преобразования.

В окне **Transformation** утилиты XML Mapper на вкладке **Mapping**, выберем пункт **Datapacket to XML**. Нажмем кнопку **Create and Test Transformation**. Мы только что создали обратное преобразование из пакета данных в XML-файл. На экране откроется окно, содержащее схему структуры XML-файла. Сохраним файл этого преобразования (команда **File | Save | Transformation**).

Теперь у нас есть два файла преобразования: из XML-документа в пакет данных и обратно, и мы можем протестировать их работу при помощи простого приложения. На этот раз, для разнообразия, воспользуемся Delphi Language IDE.

Создадим в IDE новый проект и разместим в его форме компоненты `XMLTransformProvider`, `ClientDataSet`, `DataSource`, `DBGid` и `DBNavigator`. Первые три компонента мы найдем на странице **Data Access** палитры компонентов, а последние два — на странице **Data Controls**.

Свойству XMLDataFile объекта XMLTransformProvider1 присвоим ссылку на файл catalog.xml. У компонента XMLTransformProvider есть два вложенных "подкомпонента" — TransformRead и TransformWrite. Первый из этих компонентов отвечает за чтение данных из XML-документа, второй — за запись. Раскроем список свойств подкомпонента TransformRead и назначим свойству TransformationFile ссылку на созданный файл преобразования из формата XML в пакет данных. Свойству TransformationFile компонента TransformWrite назначим ссылку на файл преобразования из пакета данных в формат XML.

Подробное описание компонентов ClientDataSet, DataSource, DBGid и DBNavigator будет дано в следующих разделах этой главы, сейчас же нам следует просто установить между этими компонентами необходимые связи. Свойству ProviderName объекта ClientDataSet1 назначим ссылку на объект XMLTransformProvider1, свойству DataSet объекта DataSource1 — ссылку на объект ClientDataSet1. Свойства DataSource объектов DBGid1 и DBNavigator1 должны ссылаться на объект DataSource1.

Схема связей между объектами представлена на рис. 15.8. Кроме компонента DBNavigator, позволяющего осуществлять навигацию по базе данных, нам понадобится кнопка, при нажатии на которую производится внесение изменений из набора данных в первичный источник данных. Добавим в форму нашего приложения кнопку **Обновить**, обработчик которой вносит изменения в XML-файл.

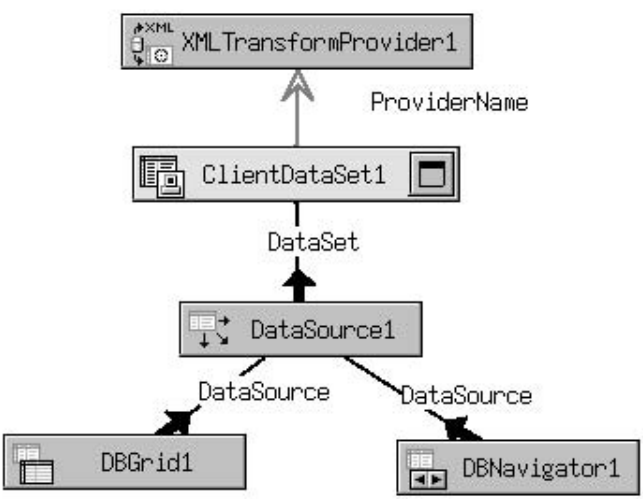


Рис. 15.8. Схема связей между компонентами приложения, работающего с XML-данными

Рассмотрим обработчики событий, которые нам нужно добавить в приложение. В листинге 15.14. представлен исходный текст обработчика события `OnCreate` объекта-формы.

Листинг 15.14. Обработчик события `OnCreate`

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ClientDataSet1.Open;
  ClientDataSet1.Last;
  MaxId := ClientDataSet1.FieldByName('id').AsInteger;
end;
```

В этом обработчике мы сначала открываем клиентский набор данных. Каждый раз, когда пользователь создает новую запись, ему необходимо ввести численный идентификатор. Само значение идентификатора особой роли не играет, главное, чтобы он не совпадал с уже существующими. Было бы удобно сделать так, чтобы при создании новой записи поле `id` заполнялось автоматически. Для этого мы вводим переменную `MaxId`, которая при запуске программы должна содержать максимальное значение идентификатора из уже существующих. При добавлении новой записи в базу программа будет автоматически присваивать полю `id` значение `MaxId`, предварительно увеличенное на единицу. Таким образом, обеспечим индивидуальность идентификаторов. Получить максимальное значение идентификатора очень просто. Поскольку сортировка записей выполняется по значению идентификатора, последняя запись и будет иметь максимальное значение. С помощью метода `Last` объекта `ClientDataSet1` мы делаем последнюю запись текущей и присваиваем переменной `MaxId` значение поля `id` этой записи.

Для того чтобы выполнить заполнение поля записи сразу после ее создания, мы назначим обработчик события `OnAfterInsert` объекту `ClientDataSet1`. Текст обработчика приводится в листинге 15.15.

Листинг 15.15. Обработчик события `OnAfterInsert` объекта `ClientDataSet1`

```
procedure TForm1.ClientDataSet1AfterInsert(DataSet: TDataSet);
begin
  Inc(MaxId);
  ClientDataSet1.FieldByName('id').AsInteger := MaxId;
end;
```

Обработчик события `OnClick` для кнопки **Обновить** приводится в листинге 15.16.

Листинг 15.16. Внесение изменений из набора данных в XML-файл

```
procedure TForm1.UpdateButtonClick(Sender: TObject);
begin
  ClientDataSet1.ApplyUpdates(-1);
end;
```

В этом обработчике мы используем метод `ApplyUpdates` объекта `ClientDataSet1`. Метод `ApplyUpdates` вносит все изменения, произведенные в наборе данных, в базу данных. Единственный его параметр - максимальное число ошибок, которое может быть допущено при внесении изменений. Если число возникших ошибок превышает это значение, операция обновления прерывается. Значение −1 соответствует неограниченному числу допустимых ошибок. Число ошибок, возникших в процессе обновления данных, возвращается методом `ApplyUpdates` в качестве результирующего значения. При завершении работы приложения следует закрыть открытый набор данных. Лучше всего сделать это в обработчике события `OnClose` главной формы.

Описывая метод `ApplyUpdates`, мы говорили о внесении изменений в базу данных. На самом деле хранилищем данных в нашем случае является XML-файл, но назначение компонента `XMLTransformProvider` как раз и заключается в том, что этот компонент позволяет работать с XML-документом как с базой данных.

Итак, у нас есть приложение, позволяющее работать с XML-документом как с базой данных (рис. 15.9). Если мы внесем новые записи, а потом просмотрим XML-документ, то увидим, что изменения в документ внесены в соответствии с его структурой. Следует особо подчеркнуть разницу между работой с XML-документом произвольной структуры как с базой данных с помощью компонента `XMLTrasformProvider` и сохранением набора данных в формате XML с помощью метода `SaveToFile` объект-клиентского набора данных, который сохраняет пакет данных в специальном формате.

Компонент *XMLTransformClient*

В предыдущем примере мы рассмотрели приложение, соответствующее локальной архитектуре приложений баз данных и использующее XML-документ в качестве хранилища информации. Еще один компонент, напрямую связанный с преобразованиями XML-формата и пакетов данных, — это компонент `XMLTransformClient`, который позволяет преобразовывать пакеты данных в произвольный формат XML. Как и в предыдущем случае, функции этого пакета не следует путать с возможностью сохранения пакета данных из клиентского набора. Повторим, что пакет данных, сохраненный в формате XML, имеет строго определенную структуру, тогда как компонент `XMLTransformClient` позволяет сохранять данные БД в *произвольном*

XML-формате. Кроме того, возможности компонента XMLTransformClient, отнюдь не ограничиваются преобразованием пакетов данных в формат XML. XMLTransformClient может выполнять и обратную функцию — вносить изменения в базу данных на основе информации, полученной из произвольных XML-документов.

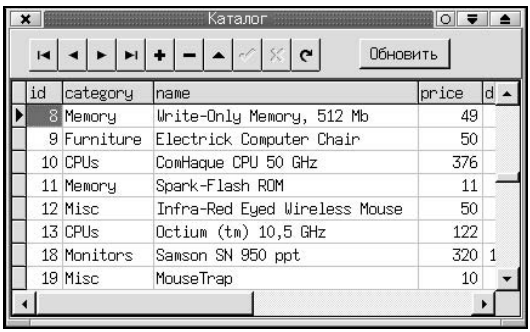


Рис. 15.9. Приложение, использующее XML-документ в качестве базы данных

Рассмотрим пример, демонстрирующий некоторые возможности компонента XMLTransformClient. Напишем приложение, преобразующее в формат XML содержимое таблицы idpricelist из базы данных, созданной в примере из предыдущей главы *(см. разд. "Компонент SQLDataSet" гл. 14,* листинг 14.6*)*. В рамках того примера мы создали приложение, заполняющее таблицу данными из XML-документа с использованием низкоуровневых возможностей компонента SQLDataSet. В данном случае мы решим обратную задачу с помощью компонента XMLTransformClient.

Создадим новое приложение в Kylix Delphi IDE и разместим в его форме компоненты SQLConnection, SQLDataSet и DataSetProvider. Настроим объект SQLConnection1 на связь с базой данных, содержащей таблицу idpricelist, и свяжем объекты SQLConnection1, SQLDataSet1 и DataSetProvider1 так, как это делалось в главе 14. В этой цепочке далее следовал компонент ClientDataSet, но теперь вместо него мы используем компонент XMLTransformClient. Поместим этот компонент в форму приложения. У полученного объекта XMLTransformClient1 тоже есть свойство ProviderName, которому следует назначить ссылку на объект DataSetProvider1.

Для того чтобы объект XMLTransformClient1 мог преобразовывать содержимое таблицы idpricelist в XML-документ, необходимо создать соответствующий файл преобразований. Создание такого файла мы начнем с разработки заготовки XML-файла, описывающего структуру того XML-документа, в который будут помещены данные. Исходный текст заготовки приводится в листинге 15.17.

Листинг 15.17. Заготовка XML-документа

```
<catalog>
<item>
<ID></ID>
<CATEGORY></CATEGORY>
<NAME></NAME>
<PRICE></PRICE>
<DISCOUNT></DISCOUNT>
</item>
<item>
<ID></ID>
<CATEGORY></CATEGORY>
<NAME></NAME>
<PRICE></PRICE>
<DISCOUNT></DISCOUNT>
</item>
</catalog>
```

Элемент `item` повторяется в этой заготовке два раза, чтобы преобразователь XML Mapper "понял", что таких элементов может быть несколько. Сохраните заготовку в файле template.xml.

Запустим утилиту XML Mapper и откроем файл template.xml. Перенесем подэлементы элемента `item` в окно **Transformations**, как было описано выше. На вкладке **Node Properites** укажем тип каждого элемента в соответствии с типом одноименного поля в таблице `idpricelist`. Выберем команду меню **Create | Datapacket from XML**. В группе **Transform Direction** выберем пункт **Datapacket to XML** и нажмем кнопку **Create and Test Transformation**. Сохраним созданный файл преобразования (команда меню **File | Save | Transformation**) под именем dp2xml.xtr. Теперь для преобразования данных из таблицы `idpricelist` в формат XML нам достаточно вызвать два метода объекта `XMLTransformClient1` (листинг 15.18).

Листинг 15.18 Преобразование пакета данных в XML-документ

```
procedure TForm1.Button2Click(Sender: TObject);
var
  XMLData : String;
  F : System.Text;
begin
  XMLTransformClient1.SetParams('','');
```

```
  XMLData := XMLTransformClient1.GetDataAsXml('dp2xml.xtr');
  System.Assign(F, 'result.xml');
  System.Rewrite(F);
  Write(F, XMLData);
  System.Close(F);
end;
```

Можно догадаться, что эта процедура является обработчиком события `OnClick` — некой кнопки, нажатие которой выполняет преобразование. В результате выполнения этой процедуры будет создан файл result.xml, содержащий данные из таблицы `idpricelist`, в формате, заданном в файле template.xml. Ключевым методом в этой процедуре является метод `GetDataAsXml`, возвращающий строку, содержащую данные из БД в заданном XML-формате. В качестве аргумента этому методу передается имя файла преобразований.

Итак, мы решили проблему переноса данных из БД в XML-документ. А можно ли с помощью компонента `XMLTransformClient` решить обратную задачу, аналогичную той, которую мы решали в главе 14, — перенести данные из XML-документа в БД? Для этого нам потребуется другой файл преобразований (и другие методы компонента `XMLTransformClient`).

Пусть у нас есть файл (назовем его newdata.xml), содержащий данные в формате, аналогичном приведенному в листинге 14.5. Внесем эти данные в таблицу `idpricelist` при помощи компонента `XMLTransformClient`.

Примечание

Таблица `idpricelist` содержит поле `id`, значения которого должны быть уникальными. Попытка вставить в таблицу записи с повторяющимися значениями `id` приведет к ошибкам. Если мы хотим добавить данные из файла, который мы уже использовали для этой цели, надо изменить, по крайней мере, значения полей `id`.

Откроем файл newdata.xml в программе XML Mapper. Выполним все описанные выше действия по определению полей в окне **Transformation** и дадим команду **Create | Datapacket from XML**. После создания пакета данных установим направление преобразования **XML to Datapacket**, а в группе **Create Datapacket as** выберем пункт **Insert Delta**. Нажмем кнопку **Create and Test Transformation**. В результате мы создали файл преобразований, конвертирующий данные из выбранного XML-формата в "дельта-пакет", позволяющий добавлять записи в базы данных. Сохраним файл преобразований под именем xml2delta.xtr.

Далее мы можем воспользоваться приложением, которое было создано для предыдущего примера. Добавим в приложение еще одну кнопку, для которой определим обработчик `OnClick` (листинг 15.19).

Листинг 15.19. Добавление записей из XML-документа в базу данных

```
procedure TForm1.Button1Click(Sender: TObject);
var
  F : file of Byte;
  S : String;
begin
  System.Assign(F, 'newdata.xml');
  System.Reset(F);
  SetLength(S, FileSize(F));
  BlockRead(F, S[1], Length(S));
  System.Close(F);
  XMLTransformClient1.ApplyUpdates(S, 'xml2delta.xtr', -1);
end;
```

Не очень корректный способ чтения данных из файла связан с тем, что с одной стороны, нам нужно, чтобы XML-данные содержались в строке `String`, с другой стороны, файл newdata.xml может включать символы перевода строки. Таким образом, вместо нескольких процедур `ReadLn` мы предпочли воспользоваться одной процедурой `BlockRead`.

Метод `ApplyUpdates` класса `TXMLTransformClient` подобен одноименному методу класса `TClientDataSet`. В качестве первого параметра мы передаем строку, содержащую XML-данные. Второй параметр — имя файла преобразований, а третий — максимально допустимое число ошибок при внесении данных.

Глава 16

Распределенные приложения баз данных

Эта глава посвящена вопросам создания распределенных приложений баз данных различных архитектур. Мы также рассмотрим использование баз данных в интернет-приложениях "в служебных целях" — для хранения данных о зарегистрированных пользователях и ведения журнала поступающих запросов.

Клиент-серверная архитектура приложений и режим автономной работы

Создать приложения клиент-серверной архитектуры для работы с такими СУБД, как MySQL или Informix достаточно просто. Эти СУБД изначально рассчитаны на работу в сетях. Достаточно только разрешить удаленным пользователям доступ к базам данных. Разработка приложений баз данных для таких СУБД почти не отличается от разработки локальных приложений. При настройке компонента `SQLConnection` в поле `HostName` необходимо указать имя или адрес узла, на котором расположен сервер.

Одна из особенностей работы в сетях связана с тем, что соединение с удаленным сервером баз данных может быть не всегда доступно. По этой причине многие клиентские программы включают возможность работы в автономном режиме или "режиме портфеля" (briefcase mode). Суть режима автономной работы заключается в том, что пользователь сохраняет на локальной машине копию таблицы, с которой он работает, редактирует ее, а затем, при подключении к удаленному серверу баз данных, вносит изменения в БД.

Для работы в автономном режиме можно использовать рассмотренный в предыдущей главе механизм MyBase. Мы можем в любой момент сохранить содержимое клиентского набора данных, подключенного к БД, с помощью метода `SaveToFile`. Затем в автономном режиме мы можем загрузить сохраненный набор данных с помощью метода `LoadFromFile`. Одна из особенностей работы с локальными файлами, содержащими наборы данных, заключается в том, что пакет данных `Delta`, содержащий информацию об изменениях, внесенных в набор данных, загружается и сохраняется при ка-

ждой загрузке/сохранении набора в файле, накапливая при этом информацию о внесенных пользователем изменениях. Этот пакет будет существовать в локальном файле с момента внесения первых изменений в сохраненный набор данных и до тех пор, пока программа не "опустошит" его явным образом, например, вызвав метод `MergeChangeLog` объекта, реализующего клиентский набор данных.

Как же вносить измененные наборы данных в исходную базу данных? На наш взгляд, наиболее подходящим средством для этого является метод `ApplyUpdates`, но не клиентского набора данных, а компонента-провайдера. Рассмотрим объявления двух перегруженных вариантов метода `ApplyUpdates` для провайдера:

```
function ApplyUpdates(const Delta: OleVariant; MaxErrors: Integer;
out ErrorCount: Integer): OleVariant; overload;
function ApplyUpdates(const Delta: OleVariant; MaxErrors: Integer;
out ErrorCount: Integer; var OwnerData : OleVariant):
OleVariant; overload;
```

Первым параметром обоих вариантов метода является переменная `Delta` типа `OleVariant`, которая служит для передачи провайдеру пакета данных `Delta`. Параметр `MaxErrors` имеет тот же смысл, что и одноименный параметр метода `ApplyUpdates` класса `TDataSet` (максимально допустимое число ошибок в процессе внесения обновлений). Параметр `ErrorCount` возвращает число возникших ошибок. Параметр `OwnerData` второго варианта метода служит для передачи произвольных данных событиям, вызываемым методом `ApplyUpdates` (`BeforeApplyUpdates` и `AfterApplyUpdates`).

Оба варианта метода `ApplyUpdates` возвращают значение типа `OleVariant`, представляющее собой пакет записей, которые не удалось внести в базу данных.

Итак, используя метод `ApplyUpdates` компонента-провайдера, мы можем передать провайдеру пакет данных `Delta`, хранящий изменения, сделанные в локальном наборе данных, с момента перехода в автономный режим. Общая схема внесения изменений в базу данных представлена в листинге 16.1.

Листинг 16.1. Переход в автономный режим и внесение изменений в базу данных

```
var
  Errc : Integer; // переменная, возвращающая число ошибок
...
// Сохраняем данные и переходим в автономный режим работы
ClientDataSet1.SaveToFile('data.cds');
...
```

```
// Вносим измененный набор данных в базу данных и подключаем набор данных
// к провайдеру
ClientDataSet1.LoadFromFile('data.cds');
if ClientDataSet1.ChangeCount > 0 then
DataSetProvider1.ApplyUpdates(ClientDataSet1.Delta, -1, Errc);
ClientDataSet1.Close;
ClientDataSet1.ProviderName := 'DataSetProvider1';
ClientDataSet1.Open;
```

Свойство `ChangeCount` содержит количество изменений, внесенных в набор данных. Мы используем это свойство для проверки наличия записей в пакете `Delta`, так как в случае, если этот пакет не содержит данных (то есть в автономном режиме изменения в набор данных не вносились), метод `ApplyUpdates` вызовет исключительную ситуацию.

Многоуровневая архитектура приложений баз данных

Kylix позволяет создавать многоуровневые (multi-tiered) приложения баз данных (в том числе кроссплатформенные) на основе протокола SOAP (Simple Object Access Protocol, простой протокол доступа к объектам). Структура многоуровневого приложения показана на схеме рис. 16.1.

Процесс передачи данных в многоуровневом приложении можно разделить на следующие этапы:

1. Клиентская программа посылает SOAP-запрос серверу.
2. SOAP-сервер подключается к СУБД и посылает запрос на получение данных. Из этих данных сервер формирует пакет данных, упаковывает его в пакет протокола SOAP и отсылает клиентской программе.
3. Клиентская программа распаковывает полученный SOAP-пакет, вносит пакет данных в локальный набор данных и отображает данные в соответствующих компонентах пользовательского интерфейса (например, в компоненте `DBGrid`).
4. Пользователь вносит изменения в локальный набор данных (удаляет, изменяет или добавляет записи). Изменения сохраняются клиентской программой в пакете `Delta`.
5. В ответ на команду обновления базы данных, например, вызов метода `ApplyUpdates`, клиентская программа посылает пакет `Delta` серверу, используя протокол SOAP.

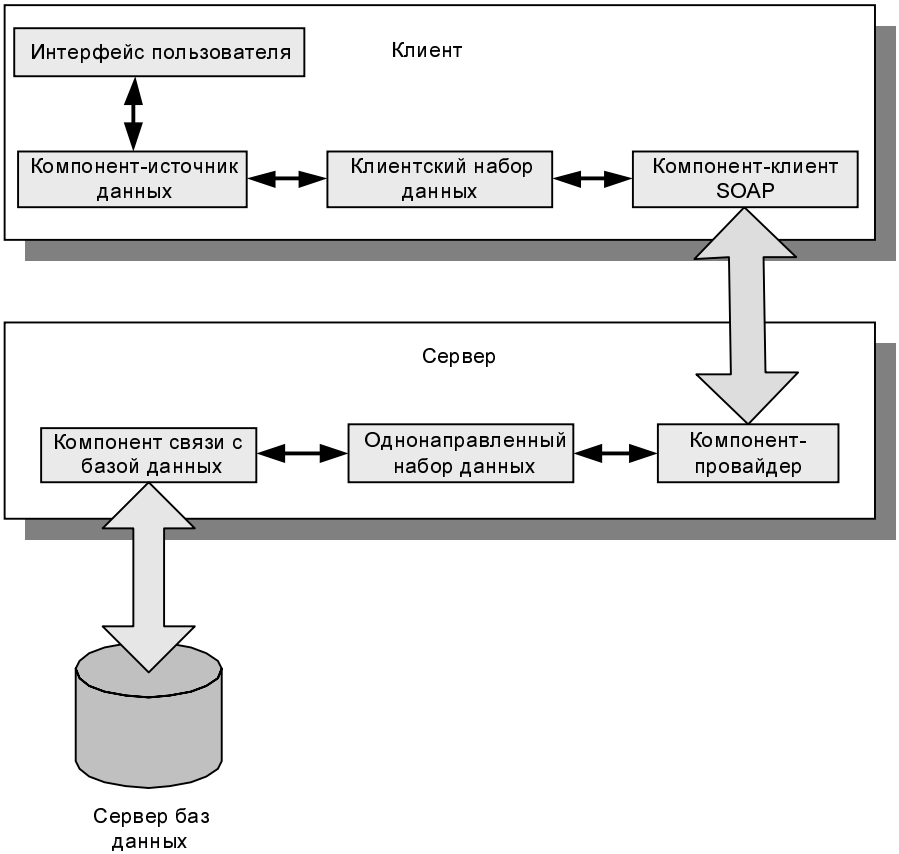


Рис. 16.1. Структура многоуровневого приложения

6. Сервер приложений получает пакет SOAP и формирует набор команд для сервера баз данных, вносящих изменения в соответствующие таблицы. Если при этом возникает какая-либо нештатная ситуация, например, запись, удаляемая одним пользователем, была перед этим изменена другим, сервер пытается согласовать содержимое клиентского набора данных с содержимым серверного набора данных, формируя набор записей, которые не удалось внести в базу данных, и отправляя его клиенту.
7. Если клиентская программа получила пакет записей, которые сервер не смог внести в базу данных, она может проинформировать об этом пользователя.
8. Выполняется обновление клиентского набора данных.

На практике многие шаги этого процесса осуществляются посредством вызова методов SOAP-интерфейсов. Ниже мы увидим, как это происходит.

Напишем на языке Delphi Language многоуровневое приложение, позволяющее пользователю просматривать и редактировать содержимое таблицы `idpricelist` *(см. гл. 14)*.

Начнем с приложения-сервера. Поскольку для передачи данных в многоуровневых приложениях Kylix используется протокол SOAP, наше приложение-сервер должно быть SOAP-сервером. На вкладке **WebServices** диалогового окна **New Items** выберем пункт **SOAP Server Application**. Будет создан главный модуль приложения-сервера и предложено создать интерфейс для него. Мы разместим компоненты сервера в отдельном модуле данных, так что от создания интерфейса на этом этапе мы откажемся. Теперь создадим новый модуль данных. На той же странице **WebServices** выбираем пункт **SOAP Server Data Module**.

В диалоговом окне **SOAP Module Wizard** укажем имя нового модуля, например, `Catalog`. В окне нового модуля (рис. 16.2) разместим компоненты `SQLConnection`, `SQLDataSet` и `DataSetProvider`.

С помощью редактора соединений настроим объект `SQLConnection1` для связи с базой данных, содержащей таблицу `idpricelist`. Установим свойство `Connected` объекта `SQLConnection1`, равным `true`. Свойству `SQLConnection` объекта `SQLDataSet1` назначим ссылку на объект `SQLConnection1`, а свойству `DataSet` объекта `DataSetProvider1` – ссылку на объект `SQLDataSet1`. Свойству `CommandText` объекта `SQLDataSet1` назначим команду:

```
select * from idpricelist
```

и присвоим значение `true` свойству `Active` этого объекта. Убедимся, что свойство `Exported` объекта `DataSetProvider1` установлено, равным `true`.

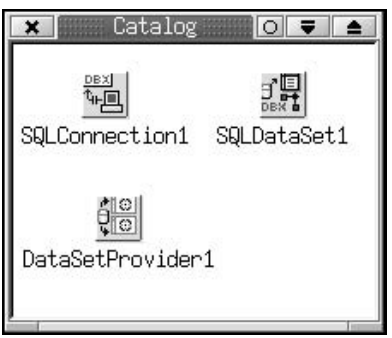


Рис. 16.2. Модуль данных приложения-сервера

На этом процесс создания простого сервера многоуровневого приложения завершен. Сохраним проект под именем `dbdemo`, скомпилируем его и скопируем получившийся двоичный файл в каталог, соответствующий типу созданного нами серверного приложения.

Приложение-клиент многоуровневой системы, как и любой клиент SOAP, может быть создано на основе заготовки стандартного приложения Kylix. Из рис. 16.1 следует, что клиентское приложение обменивается данными с сервером при помощи компонента-клиента SOAP. В качестве такового используется компонент `SOAPConnection` (класс `TSOAPConnection`), расположенный на вкладке **WebServices** палитры компонентов. Поместим этот компонент в форму приложения. В инспекторе объектов назначим свойству `URL` объекта `TSOAPConnection1` сетевой адрес SOAP-сервиса, предоставляемого приложением сервером. Например, если сервер выполнен в виде CGI-приложения, этот адрес может выглядеть так: **http://localhost/cgi-bin/dbdemo/SOAP/**. Теперь установим свойство `UseSOAPAdapter`, равным `false`. Чем управляет свойство `UseSOAPAdapter`? Компонент `SOAPConnection` позволяет клиентскому приложению получить доступ к двум интерфейсам, экспортируемым приложением-сервером: `IAppServer` и `IAppServerSOAP`. Разница между этими двумя интерфейсами заключается в основном в том, что второй, введенный в более поздних релизах Kylix 2, использует формат вызова метода `safecall`. Приложения, созданные в Kylix 3, поддерживают оба интерфейса, и если мы наберем в браузере ссылку **http://localhost/cgi-bin/dbdemo**, то увидим, что приложение-сервер экспортирует как интерфейс `IAppServer`, так и интерфейс `IAppServerSOAP`. Однако здесь есть одна тонкость. Взглянем на объявление интерфейса `ICatalog` и класса `TCatalog` в исходном тексте приложения сервера (листинг 16.2).

Листинг 16.2. Объявление интерфейса `ICatalog`

```
ICatalog = interface(IAppServer)
    ['{421DB7DB-0FD0-D611-94DC-00C0DF0894B8}']
  end;

  TCatalog = class(TSoapDataModule, ICatalog, IAppServer)
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
    DataSetProvider1: TDataSetProvider;
  private

  public

  end;
```

Как видим, и интерфейс, и реализующий его методы класс основаны на интерфейсе IAppServer, и потому, хотя интерфейс IAppServerSOAP и публикуется, попытка обратиться к его методам в программе-клиенте приведет к ошибкам. Свойство UseSOAPAdapter компонента SOAPConnection определяет, к какому из экспортируемых сервером интерфейсов клиент обращается по умолчанию. Если значение этого свойства равно true, клиент будет обращаться к интерфейсу IAppServerSOAP, в противном случае — к интерфейсу IAppServer. Поскольку наш сервер реализует методы интерфейса IAppServer, мы устанавливаем значение свойства UseSOAPAdapter, равным false. Если мы хотим использовать интерфейс IAppServerSOAP, внесем соответствующие изменения в объявление интерфейса ICatalog и класса TCatalog. Объявление интерфейса IAppServer расположено в модуле Midas (файл Midas.hpp для языка C++), а объявление интерфейса IAppServerSOAP — в модуле SOAPMidas (файл SOAPMidas.hpp).

Если мы пишем приложение-клиент для связи с сервером, созданным в среде Delphi 6 или первых релизах Kylix 2, мы можем использовать только интерфейс IAppServer.

Теперь мы можем установить свойство Connected объекта SOAPConnection1 равным true. Ничего особенного при этом не происходит. Свойство Connected — просто флажок, обозначающий, что объект готов отправить запрос серверу, когда это будет необходимо.

Разместим в форме приложения компонент ClientDataSet. Свойству RemoteServer соответствующего объекта назначим ссылку на объект SOAPConnection1. После этого свойство ProviderName объекта ClientDataSet1 примет вид раскрывающегося списка, в котором будут перечислены все имена объектов-провайдеров приложения-сервера. Компонент ClientDataSet "узнает" имена с помощью метода AS_GetProviderNames интерфейса IAppServer (или SAS_GetProviderNames интерфейса IAppServerSOAP). Эти методы возвращают вариантный массив, содержащий список имен объектов-провайдеров, зарегистрированных в приложении-сервере. Получить ссылку на интерфейсы IAppServer и IAppServerSOAP можно при помощи методов компонента SOAPConnection GetServer и GetSoapServer соответственно.

Установив имя удаленного провайдера, мы можем присвоить значение true свойству Active объекта, реализующего клиентский набор данных. Далее мы можем связать объект ClientDataSet1 с объектом-источником данных (компонент DataSource), который, в свою очередь, можно связать с различными компонентами, реализующими пользовательский интерфейс (DBGrid и DBNavigator). Внешний вид формы приложения во время разработки приводится на рис. 16.3.

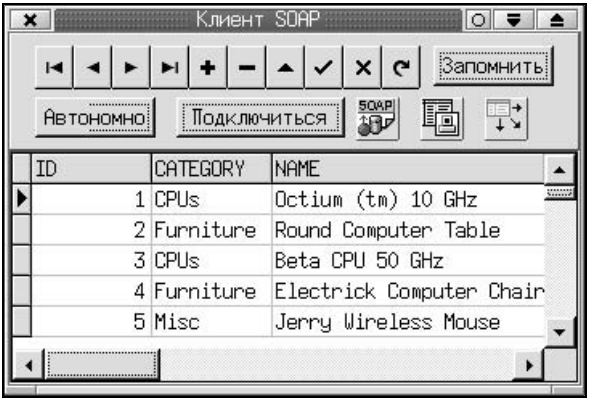


Рис. 16.3. Форма клиентского приложения во время разработки

Рассмотрим исходный текст приложения клиента (листинг 16.3).

Листинг 16.3. Исходный текст приложения-клиента многоуровневой архитектуры

```
unit ClientUnit1;

interface

uses
  SysUtils, Types, Classes, Variants, QTypes, QGraphics, QControls,
 QForms, QDialogs, QStdCtrls, DB, DBClient, SOAPConn, QGrids, QDBGrids,
 Midas, Rio, SOAPHTTPClient, QExtCtrls, QDBCtrls;

type
  TForm1 = class(TForm)
    SOAPConnection1: TSOAPConnection;
    ClientDataSet1: TClientDataSet;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    UpdateButton: TButton;
    DBNavigator1: TDBNavigator;
    Panel1: TPanel;
    DisconnectButton: TButton;
    ConnectButton: TButton;
```

```
    procedure UpdateButtonClick(Sender: TObject);
    procedure DisconnectButtonClick(Sender: TObject);
    procedure ConnectButtonClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}

procedure TForm1.UpdateButtonClick(Sender: TObject);
begin
  if SOAPConnection1.Connected then
  ClientDataSet1.ApplyUpdates(-1)
  else
  ClientDataSet1.SaveToFile('local.cds');
end;

procedure TForm1.DisconnectButtonClick(Sender: TObject);
begin
  ClientDataSet1.SaveToFile('local.cds');
  SOAPConnection1.Connected := False;
end;

procedure TForm1.ConnectButtonClick(Sender: TObject);
var
  ServerIntf : IAppServer;
  Errc : Integer;
  Data : OleVariant;
begin
  SOAPConnection1.Connected := True;
```

```
  ServerIntf := SOAPConnection1.GetServer;
  if not ClientDataSet1.Active then
  begin
    if FileExists('local.cds') then
    ClientDataSet1.LoadFromFile('local.cds')
    else ClientDataSet1.Open;
  end;
  if ClientDataSet1.ChangeCount > 0 then
  ServerIntf.AS_ApplyUpdates(ClientDataSet1.ProviderName,
  ClientDataSet1.Delta, -1, Errc, Data);
end;


end.
```

Наше приложение позволяет работать с данными в автономном режиме. Для перехода в автономный режим служит кнопка **Автономно** (объект `DisconnectButton`). Обработчик события `OnClick` этого объекта сохраняет содержимое клиентского набора данных в файле local.cds и устанавливает значение свойства `Connected` объекта `SOAPConnection1`, равным `false`.

Кнопка **Запомнить** (объект `UpdateButton`) служит для внесения изменений в источник данных. При работе в автономном режиме нельзя обращаться к методу `ApllyUpdates`, поэтому в обработчике `UpdateButtonClick` мы проверяем значение свойства `Connected` объекта `SOAPConnection1`, и, если это значение равно `true`, вызываем метод `ApllyUpdates` объекта `ClientDataSet1`, в противном случае, вызываем метод `SaveToFile`, который сохраняет не только содержимое набора данных, но и текущее содержимое пакета `Delta`. В процессе автономной работы метод `SaveToFile` можно вызывать сколько угодно раз, при этом все внесенные в набор данных изменения будут последовательно накапливаться в сохраняемом пакете `Delta`.

Кнопка **Подключиться** (объект `ConnectButton`) служит для подключения клиента к серверу и внесения изменений, сделанных в автономном режиме, в базу данных. В обработчике `ConnectButtonClick` мы присваиваем свойству `Connected` объекта `SOAPConnection1` значение `true`. Далее мы получаем ссылку на интерфейс `IAppServer`. Затем, если клиентский набор данных не открыт, мы либо загружаем данные из локального файла (если этот файл существует), либо просто открываем набор данных. Изменения в базу данных вносятся при помощи метода `ApplyUpdates` интерфейса `IAppServer`, который выступает как "представитель" серверного объекта-провайдера.

Использование баз данных в Web-приложениях

В этом разделе мы напишем Web-приложение, требующее регистрации пользователей для доступа к своим сервисам и использующее базы данных для ведения журнала запросов и списка учетных записей пользователей. Приложение, описанное в этом разделе, послужит также основой для распределенных приложений интернет-архитектуры, описанных в следующем разделе.

Создадим две новые таблицы. Одну назовем `accounts`, она будет содержать учетные записи пользователей, другую — `logs`, в ней серверное приложение будет вести журнал запросов.

Таблица `accounts` должна содержать следующие поля:

- `id` — целочисленное поле, являющееся первичным ключом и служащее для идентификации учетной записи пользователя;
- `user_name` — поле строкового типа, содержащее имя пользователя для регистрации;
- `user_pwd` – строковое поле, содержащее пароль пользователя;
- `first_name` — строковое поле, содержащее "реальное" имя пользователя;
- `surname` — строковое поле, содержащее фамилию пользователя.

Поля таблицы `logs` следующие:

- `date_str` — содержит данные о времени запроса в строковом формате;
- `user_id` — целочисленный идентификатор учетной записи пользователя, пославшего запрос;
- `rem_addr` — строковое поле, содержащее IP-адрес пользователя;
- `page_req` — строковое поле, содержащее имя запрошенной пользователем страницы (CGI-сервиса);
- `result_code` — целочисленное поле, содержащее HTTP-код результата выполнения запроса.

В таблицу `accounts` внесем учетную запись с идентификатором 0 и именем пользователя `unknown` (остальные поля записи оставим пустыми). Эта запись будет служить для идентификации неизвестных (не зарегистрировавшихся) пользователей.

Поскольку серверное приложение будет искать записи пользователей по значению поля `user_name`, для ускорения поиска имеет смысл создать в базе данных индекс для этого поля:

```
create index userind on accounts(user_name);
```

Создадим генератор для автоматической генерации значений поля `id` и присвоим ему начальное значение:

```
create generator idgen;
set generator idgen to 0;
```

Теперь создадим заготовку CGI-проекта WebBroker (пункт **Web Server Application** страницы **New** диалогового окна **New Items**). В форме Web-модуля разместим компоненты `SQLConnection`, `SQLDataSet` и `PageProducer` (рис. 16.4).

Рис. 16.4. Форма Web-модуля

Объект `SQLClientDataSet` назван здесь `AccountsDataSet`. Мы не можем использовать однонаправленный набор данных, так как нам потребуется выполнять поиск в наборе учетных записей. Объект `SQLConnection1` должен быть настроен на работу с базой данных, содержащей таблицы `accounts` и `logs`. Назначим свойству `Connected` объекта `SQLConnection1` значение `true`. Свойству `CommandText` объекта `AccountsDataSet` следует присвоить строку, содержащую параметрический запрос:

```
select * from accounts where user_name := UserName
```

Описанная выше индексация поля `user_name` должна ускорить выполнение этой команды.

Далее в редакторе свойства `Actions` Web-модуля создадим объекты-сервисы (actions) CGI-приложения. Значения свойств объектов перечислены в табл. 16.1.

Таблица 16.1. *Значения свойств объектов-сервисов CGI*

Name	PathInfo	Default
Default	/	True
Register	/register	False

Таблица 16.1 *(окончание)*

Name	PathInfo	Default
Login	/login	False
Logout	/logout	False
Access	/access	False

Сервис `Default` предоставляется CGI-приложением по умолчанию. Если пользователь еще не зарегистрировался, этот сервис перенаправляет его на сервис `Login`, который предоставляет страницу, содержащую форму для ввода имени пользователя и пароля, и обрабатывает данные, переданные из формы. Страница, предоставляемая сервисом `Login`, содержит также ссылку на сервис `Register`, перейдя по которой, пользователь может создать новую учетную запись на сервере. Сервис `Register` не только предоставляет страницу, содержащую форму для создания новой учетной записи, но и обрабатывает данные, поступившие с этой страницы.

Если пользователь уже ввел имя и пароль, соответствующие существующей учетной записи, сервис `Default` перенаправляет его на страницу `Access`, которая доступна только зарегистрированным пользователям.

Сервис `Logout` позволяет пользователю выйти из режима регистрации на сервере. Информация о зарегистрированных в данный момент пользователях передается при помощи cookie *(подробную информацию о технологии cookies см. в гл. 7)*. После того, как пользователь ввел допустимые имя и пароль или создал новую учетную запись, приложение посылает клиентской программе (браузеру) cookie с именем, соответствующим имени пользователя и значением, соответствующим его паролю. Поскольку браузер сохраняет поступившие cookie, пользователь может захотеть удалить их по окончании работы, чтобы другие пользователи, работающие с той же программой, не воспользовались его реквизитами. Сервис `Logout` посылает клиенту тот же cookie, но с пустым значением, так что после этого для получения доступа к странице `access` снова понадобится регистрация на сервере.

Рассмотрим тексты HTML-страниц, используемые создаваемым приложением (листинги 16.4, 16.5, 16.6).

Листинг 16.4. Страница login.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>Регистрация</TITLE>
  </HEAD>
```

```
  <BODY>
      <P>Используйте следующую форму для регистрации на сервере</P>
     <HR>
     <P><FORM action="<#server>/login" method="POST">
     Имя: <INPUT name="username" type="TEXT" size=8>
     <P>Пароль: <INPUT type="PASSWORD"  name="password" size=8></P>
     <P><INPUT type="SUBMIT" value="Регистрация"></P>
     </FORM></P>
     <HR>
     <P>Или создайте <A HREF="<#server>/register">учетную запись</A></P>
  </BODY>
</HTML>
```

Листинг 16.5. Страница register.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>Новая учетная запись</TITLE>
  </HEAD>
  <BODY>
    <#regstr>
     <HR>
     <P><FORM action="<#server>/register" method="POST">
     Имя: <INPUT name="name" type="TEXT" size=10>
     <P>Фамилия: <INPUT name="surname" type="TEXT" size=15></P>
     <P>Имя пользователя: <INPUT name="username" type="TEXT" size=8></P>
     <P>Пароль: <INPUT name="password" type="TEXT" size=8></P>
     <P><INPUT type="SUBMIT" value="Отправить"></P>
     </FORM></P>
     <HR>
  </BODY>
</HTML>
```

Листинг 16.6. Страница access.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>Добро пожаловать</TITLE>
```

```
  </HEAD>
  <BODY>
    <H2> Добро пожаловать, <#name>!</H2>
    Вы зарегистрированы.
    <P><A HREF="<%server>/logout">Выйти</A></P>
  </BODY>
</HTML>
```

Все страницы, используемые приложением, лучше скопировать в специально созданный каталог, например: /var/www/webdemo. Как видно из листингов, страницы используют тэги шаблонов, которые заменяются объектом `PageProducer1` на соответствующие значения. Тэг `<#server>` будет заменен на сетевой адрес серверного приложения, тэг `<#regstr>` будет заменен строкой, информирующей пользователя о необходимых действиях или ошибках, возникших при создании новой учетной записи, а вместо тэга `<#name>` программа вставит в текст страницы имя и фамилию пользователя, полученные из учетной записи.

Рассмотрим исходные тексты приложения (листинги 16.7 и 16.8):

Листинг 16.7. Заголовочный файл главного модуля (фрагмент)

```
#ifndef Unit1H
#define Unit1H
...
//---------------------------------------------------------------------------
class TWebModule1 : public TWebModule
{
  ...
  private:    // User declarations
        int userid;
        String RegisterString, FirstName, Surname;
        void UserAuth(const String UserName, const String Password);
        void SaveLog(TWebRequest * Request, TWebResponse * Response);
        void SetCookie(TWebResponse * Response, const String UserName,
const String Password);
public:       // User declarations
       __fastcall TWebModule1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TWebModule1 *WebModule1;
//---------------------------------------------------------------------------
#endif
```

Листинг 16.8. Реализация класса главного модуля

```
//---------------------------------------------------------------------
#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.xfm"

TWebModule1 *WebModule1;
//---------------------------------------------------------------------
__fastcall TWebModule1::TWebModule1(TComponent* Owner)
       : TWebModule(Owner)
{
}
//---------------------------------------------------------------------
void TWebModule1::UserAuth(const String UserName, const String Password)
{
  userid = 0;
  if ((UserName == "") || (Password == "")) return;
  AccountsDataSet->ParamByName("UserName")->Value = Variant(UserName);
  AccountsDataSet->Open();
  if (AccountsDataSet->FieldByName("USER_PWD")->AsString == Password)
  {
    userid = AccountsDataSet->FieldByName("ID")->AsInteger;
    FirstName = AccountsDataSet->FieldByName("FIRST_NAME")->AsString;
    Surname = AccountsDataSet->FieldByName("SURNAME")->AsString;
  }
  AccountsDataSet->Close();
}
void TWebModule1::SaveLog(TWebRequest * Request, TWebResponse * Response)
{
  String SQLCmd;
  String DateStr = FormatDateTime("dd/mm/yy hh:nn:ss", Now());
  String RemAddr = Request->RemoteAddr;
  String PageReq = Request->PathInfo;
  if (RemAddr == "") RemAddr = "unknown";
  if (PageReq == "") PageReq = "/";
  SQLCmd.printf("insert into logs values('%s', '%i', '%s', '%s', %i)",
```

```
  &DateStr[1], userid, &RemAddr[1], &PageReq[1], Response->StatusCode);
  SQLConnection1->Execute(SQLCmd, NULL, NULL);
}
void TWebModule1::SetCookie(TWebResponse * Response, const String
UserName, const String Password)
{
  TCookie * Cookie = Response->Cookies->Add();
  Cookie->Name = UserName;
  Cookie->Value = Password;
}
void __fastcall TWebModule1::WebModuleBeforeDispatch(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  String UserName, Password;
  userid = 0;
  if(Request->CookieFields->Count > 0)
  {
    UserName = Request->CookieFields->Names[0];
    Password = Request->CookieFields->Values[UserName];
    UserAuth(UserName, Password);
  }
 }
//---------------------------------------------------------------------
void __fastcall TWebModule1::WebModule1DefaultAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  if (userid == 0)
  Response->SendRedirect(Request->ScriptName+"/login");
  else
  Response->SendRedirect(Request->ScriptName+"/access");
}
//---------------------------------------------------------------------
void __fastcall TWebModule1::WebModule1LoginAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  if (userid != 0)
  Response->SendRedirect(Request->ScriptName + "/access");
  String UserName = Request->ContentFields->Values["username"];
```

```
  String Password = Request->ContentFields->Values["password"];
  if (UserName == "")
  {
    PageProducer1->HTMLFile = "/var/www/webdemo/login.html";
    Response->Content = PageProducer1->Content();
    return;
  }
  UserAuth(UserName, Password);
  if (userid == 0)
  {
    Response->StatusCode = 403;
    Response->Content = "Неверный пароль";
  }
  else
  {
    SetCookie(Response, UserName, Password);
    Response->SendRedirect(Request->ScriptName+"/access");
  }
}
//---------------------------------------------------------------------------
void __fastcall TWebModule1::WebModule1LogoutAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
    String UserName;
    if (Request->CookieFields->Count > 0)
    UserName = Request->CookieFields->Names[0];
    SetCookie(Response, UserName, "");
    SaveLog(Request, Response);
    Response->SendRedirect(Request->ScriptName+"/");
}
//---------------------------------------------------------------------------
void __fastcall TWebModule1::WebModule1RegisterAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  if ((Request->ContentFields->Values["username"] == "") ||
  (Request->ContentFields->Values["password"] == ""))
  {
    RegisterString = "Введите имя пользователя и пароль";
```

```
    PageProducer1->HTMLFile = "/var/www/webdemo/register.html";
    Response->Content = PageProducer1->Content();
    return;
  }
  String UserName = Request->ContentFields->Values["username"];
  String Password = Request->ContentFields->Values["password"];
  FirstName = Request->ContentFields->Values["name"];
  Surname = Request->ContentFields->Values["surname"];
  AccountsDataSet->ParamByName("UserName")->Value = Variant(UserName);
  AccountsDataSet->Open();
  if (AccountsDataSet->FieldByName("USER_NAME")->AsString != "")
  {
    RegisterString.printf("Учетная запись с именем %s уже существует",
    &UserName[1]);
    PageProducer1->HTMLFile = "/var/www/webdemo/register.html";
    Response->Content = PageProducer1->Content();
  }
  else
  {
    String SQLCmd;
    SQLCmd.printf("insert into accounts values(gen_id(idgen, 1), ↵
    '%s', '%s', '%s', '%s')",
    &UserName[1], &Password[1], &FirstName[1], &Surname[1]);
    SQLConnection1->Execute(SQLCmd, NULL, NULL);
    SetCookie(Response, UserName, Password);
    Response->SendRedirect(Request->ScriptName+"/access");
  }
}
//---------------------------------------------------------------------------
void __fastcall TWebModule1::WebModule1AccessAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  if (userid == 0)
  {
    Response->StatusCode = 403;
    Response->Content = "Неверный пароль";
  }
  else
```

```
  {
    PageProducer1->HTMLFile = "/var/www/webdemo/access.html";
    Response->Content = PageProducer1->Content();
  }
}
//---------------------------------------------------------------------
void __fastcall TWebModule1::WebModuleAfterDispatch(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  SaveLog(Request, Response);
}
//---------------------------------------------------------------------
void __fastcall TWebModule1::PageProducer1HTMLTag(TObject *Sender,
      TTag Tag, const AnsiString TagString, TStrings *TagParams,
      AnsiString &ReplaceText)
{
  if (TagString == "server")
  {
    ReplaceText = Request->ScriptName;
    return;
  }
  if (TagString == "regstr")
  {
    ReplaceText = RegisterString;
    return;
  }
  if (TagString == "name")
  {
    ReplaceText = FirstName + " " + Surname;
    return;
  }
}
//---------------------------------------------------------------------
```

Сохраним проект под именем webdemo.

Функция `UserAuth` осуществляет авторизацию пользователя. Она проверяет, не являются ли переданные ей аргументы пустыми, а затем пытается найти в таблице `accounts` запись со значением поля `user_name`, соответствующим переданному ей имени пользователя, и затем сравнивает значение поля `user_pwd` с предоставленным ей паролем. Если подходящая запись найдена,

функция присваивает переменной `userid` идентификатор учетной записи, в противном случае этой переменной присваивается значение 0. Следует отметить, что если учетной записи с указанным именем пользователя не существует, поля, возвращенные объектом `AccountsDataSet`, будут пустыми, однако функция `UserAuth` следит за тем, чтобы переданное ей значение пароля не было пустым, так что пользователь, предоставивший вместо пароля пустую строку, не пройдет авторизацию. Кроме этого функция заполняет переменные `FirstName` и `Surname`. Делается это для того, чтобы другим функциям приложения, в частности, обработчику события `OnHTMLTag` объекта `PageProducer1` не потребовалось снова обращаться к базе данных.

Функция `SaveLog` добавляет новую запись в таблицу `logs`. Необходимые значения функция получает из переданных ей параметров `Request` и `Response`. Для вставки записи в таблицу используется метод `Execute` объекта `SQLConnection1`, позволяющий выполнять SQL-команды, не возвращающие наборы записей (такие как команда `insert`). Метод `Execute` может работать с параметрическими командами, однако в нашем случае будет проще (и быстрее) сформировать текст команды с помощью метода `printf` объекта класса `String`.

Функция `SetCookie` формирует cookie, который будет передан зарегистрировавшемуся пользователю для подтверждения его реквизитов при последующих запросах.

Проверка реквизитов удаленного пользователя выполняется в функции `WebModuleBeforeDispatch`, являющейся обработчиком события `OnBeforeDispatch` Web-модуля. Благодаря этому все остальные функции программы уже "знают", является ли пользователь, пославший запрос, зарегистрированным пользователем.

Функция `WebModule1DefaultAction` является обработчиком события `OnAction` для сервиса `Default`. Эта функция осуществляет перенаправление клиента на другие страницы в зависимости от того, предоставлены ли клиентом валидные реквизиты или нет.

Функция `WebModule1LoginAction`, представляющая собой обработчик события `OnAction` сервиса `Login`, сначала проверяет, зарегистрирован ли пользователь. Если пользователь уже зарегистрирован, клиентская программа перенаправляется на страницу `/access`. Если пользователь не зарегистрирован, и запрос к сервису `Login` не содержит имени пользователя и пароля, сервис высылает клиенту страницу login.html. В противном случае выполняется авторизация пользователя, и, если пользователь прошел авторизацию, его клиентская программа перенаправляется на страницу `/access`.

Обработчик события `OnAction` сервиса `Logout` — `WebModule1LogoutAction` — "сбрасывает" реквизиты пользователя, посылая клиенту cookie с пустым значением.

Обработчик WebModule1RegisterAction, "обслуживающий" сервис Register, подобен обработчику для сервиса Login. Обработчик сначала проверяет, содержит ли направленный сервису запрос необходимые CGI-поля, и, если нет, — отправляет клиенту страницу register.html, содержащую форму для создания новой учетной записи. Если в параметрах CGI-запроса содержатся данные, необходимые для создания новой учетной записи, функция проверяет, нет ли в таблице accounts учетной записи с аналогичным именем пользователя. Если такая запись есть, клиенту снова высылается форма создания учетной записи с соответствующим пояснением. Если учетной записи с указанным именем пользователя нет, функция создает новую запись в таблице accounts методом, аналогичным тому, при помощи которого функция SaveLog записывает информацию в таблицу logs. После этого функция WebModule1RegisterAction вызывает функцию SetCookie и перенаправляет клиента на страницу /access.

Обработчик события OnAction сервиса Access (WebModule1AccessAction) проверяет, зарегистрирован ли пользователь, обращающийся к сервису, и высылает страницу access.html зарегистрированным пользователям.

Обработчик события OnAfterDispatch (WebModuleAfterDispatch) вызывает функцию SaveLog. Поскольку функция SaveLogs вызывается после обработки запроса, объект Response уже содержит статус-код результата выполнения запроса, который, наряду с другими данными, записывается в журнал. Функция SaveLog вызывается еще в одном месте программы, а именно, в обработчике WebModule1LogoutAction. Сервис Logout всегда перенаправляет клиента на другой сервис, поэтому обработчик события OnAfterDispatch для него никогда не вызывается. По этой причине, если мы хотим, чтобы в журнале отображались обращения к сервису Logout, нам следует вызвать функцию SaveLog в обработчике события этого сервиса.

Написать локальное приложение баз данных для управления таблицами accounts и logs довольно просто. С помощью такого приложения администратор системы мог бы управлять учетными записями пользователей, а также просматривать журнал запросов, используя различные критерии выборки записей (в этом и заключается преимущество хранения журналов в базах данных). Рассматривать все аспекты создания подобного локального приложения мы не будем. Остановимся лишь на одном вопросе, которого прежде не касались, а именно, на создании отношений Master/Detail между таблицами баз данных.

В таблице logs пользователь определяется идентификатором соответствующей учетной записи. При просмотре этой таблицы мы можем захотеть получить расширенные данные о пользователе, хранящиеся в таблице accounts. Система отношений Master/Detail позволяет установить связь между ними на основе соответствия полей, так что при выборе пользователем записи в одной таблице (Master), автоматически выбирается соответствующая ей запись другой таблицы (Detail).

Пусть у нас есть приложение, позволяющее работать с таблицами `accounts` и `logs`. Для работы с каждой таблицей предназначен собственный клиентский набор данных (объекты `AccountsClientDataSet` и `LogsClientDataSet`). Свойства `CmdText` этих объектов должны соответственно содержать команды:

```
select * from accounts
```

и

```
select * from logs
```

Назначим свойству `MasterSource` объекта `LogsClientDataSet` ссылку на объект `AccountsClientDataSet`. Таким образом, мы создали отношения Master/Detail между таблицами `logs` и `accounts`. Теперь нужно установить соответствие между свойствами, на основе которого будут выбираться записи. Нажмем кнопку с символом многоточия в поле свойства `MasterFields` объекта `AccountsClientDataSet`. Откроется окно редактора **Field Link Designer** (рис. 16.5), в котором можно выбрать связанные поля таблиц.

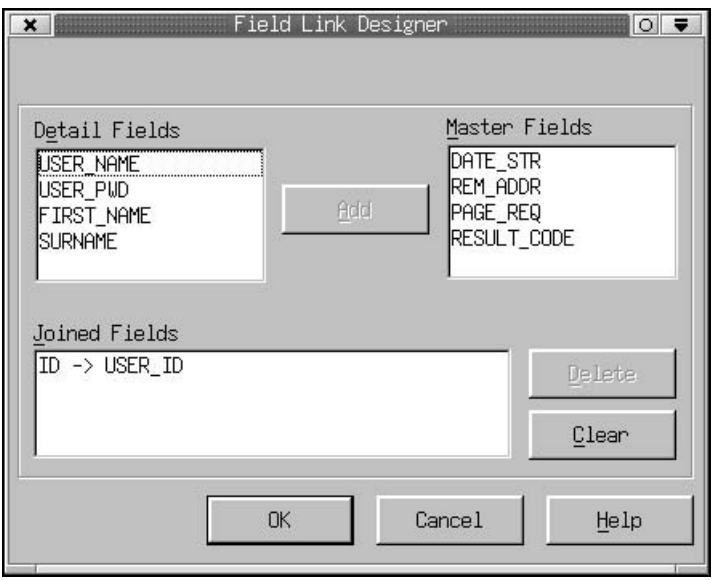


Рис. 16.5. Редактор Field Link Designer

В списке **Detail Fields** выберем поле `ID`, а в списке **Master Fields** — поле `USER_ID`, и нажмем кнопку **Add**. Нажмем кнопку **OK**. Теперь при выборе новой записи из таблицы `logs` (например, с помощью компонента `DBGrid`), содержимым набора данных `AccountsClientDataSet` станет запись, в которой значение поля `ID` совпадает со значением поля `USER_ID` выбранной записи таблицы `logs`.

Интернет-архитектура приложений баз данных

К приложениям интернет-архитектуры относятся распределенные приложения, использующие стандартные протоколы Интернета для обмена данными и Web-браузеры в качестве клиентских программ. В этом разделе мы напишем приложение для работы с базами данных при помощи технологии WebBroker (в качестве основы мы воспользуемся приложением webdemo), а также рассмотрим средства разработки интернет-приложений баз данных, предоставляемые технологией WebSnap.

Использование наборов данных в интернет-приложениях

При разработке интернет-приложений следует по возможности ограничиться использованием однонаправленных наборов данных, которые отличаются более высоким быстродействием и меньшей требовательностью к ресурсам по сравнению с клиентскими наборами. Клиентские наборы данных обычно пытаются загрузить все содержимое таблицы данных, даже если в этом нет необходимости. Напомним, что в ОС Linux мы не можем создавать серверные приложения "непрерывного действия", по крайней мере, для сервера Apache и с помощью Kylix. Новая копия приложения запускается для выполнения каждого запроса, а значит, клиентский набор, если он используется, должен каждый раз загружать таблицу.

Компоненты WebBroker и WebSnap, предназначенные для работы с базами данных, позволяют обращаться непосредственно к однонаправленным наборам данных. Если в интернет-приложении необходимо предоставить удаленному пользователю возможность добавлять записи в таблицу, это можно реализовать с помощью SQL-команд и методов компонента `SQLConnection` (как мы делали это при добавлении записей в журнал и создании новых учетных записей пользователей в приложении webdemo в предыдущем разделе). Такие операции, как индексирование полей, сортировка и отбор данных по заданному критерию, также можно выполнять на уровне сервера баз данных, посылая ему соответствующие SQL-команды.

Компоненты технологии WebBroker

Для работы с базами данных технология WebBroker предоставляет три компонента-генератора контента, наиболее интересными из которых являются компоненты `DataSetTableProducer` и `DataSetPageProducer`.

Добавьте в форму главного модуля приложения webdemo три компонента: `SQLConnection`, `SQLDataSet` и `DataSetTableProducer` (последний компонент

расположен на странице **Internet** палитры компонентов). Новый объект `SQLConnection2` настроим на работу с базой данных, содержащей таблицу `idpricelist`. Новый объект `SQLDataSet2` назовем `IdPriceListDataSet` и свяжем с объектом `SQLConnection2`. Значение свойства `CommandText` объекта `IdPriceListDataSet` устанавливать не надо, так как это будет сделано в тексте приложения. Свойству `DataSet` объекта `DataSetTableProducer1` присвоим ссылку на объект `IdPriceListDataSet`. Внешний вид формы Web-модуля показан на рис. 16.6.

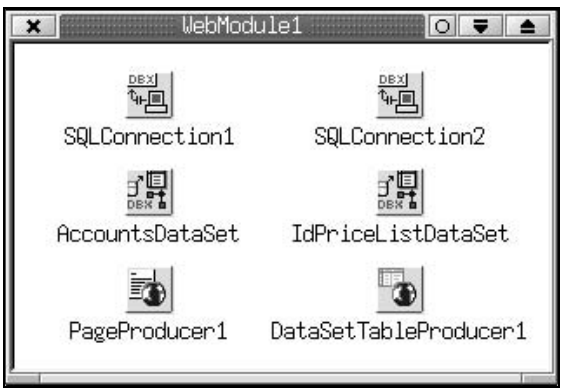


Рис. 16.6. Главная форма Web-модуля

Компонент `DataSetTableProducer` позволяет отображать содержимое таблицы базы данных в формате HTML-таблицы. При этом источником данных для компонента `DataSetTableProducer` может служить компонент, реализующий набор данных (в том числе и однонаправленный).

Компонент `DataSetTableProducer` позволяет настраивать внешний вид HTML-страницы, содержащей таблицу, в весьма широких пределах. Свойство `Caption` этого компонента позволяет установить заголовок таблицы, т. е. текст, который будет отображаться над таблицей. Свойство `Columns` является коллекцией объектов `THTMLTableColumn`, описывающих столбцы таблицы и очень похожих на объекты `TColumn` коллекции `Columns` компонента `DBGrid` (в частности, поле `Title` объекта `THTMLTableColumn` позволяет установить параметры оформления и текст заголовочной ячейки столбца). Свойства `Header` и `Footer`, представляющие собой списки строк, позволяют указать HTML-текст, расположенный, соответственно, перед и после таблицы. С помощью свойства `TableAttributes` можно установить различные параметры внешнего вида всей таблицы (ширина границ между ячейками, цвет фона, выравнивание, ширина таблицы относительно страницы), а свойство `RowAttributes` позволяет устанавливать параметры оформления строк (цвет фона, горизонтальное и вертикальное выравнивание текста). Важным является также свойство `MaxRows`, позволяющее указать макси-

мально допустимое число строк таблицы. Присвоение этому свойству значения −1 сделает максимальное число строк в таблице неограниченным.

Среди событий компонента `DataSetTableProducer` особое внимание заслуживает событие `OnFormatCell`, позволяющее настраивать внешний вид отдельных ячеек. Обработчик этого события получает полную информацию о положении ячейки в таблице, а также данные, которые компонент-генератор собирается поместить в этой ячейке (обработчик может изменить эти данные). С помощью переданных ему параметров обработчик может также изменить цвет фона и горизонтальное и вертикальное выравнивание текста в ячейке. Стоит отметить, что возможности форматирования ячеек с помощью обработчика события `OnFormatCell` гораздо шире, чем это может показаться на первый взгляд, ведь, модифицируя текст, выводимый в ячейке, обработчик может добавлять в него любые HTML-тэги, управляющие внешним видом ячейки, например, изменяющие цвет и стиль используемого шрифта.

Вернемся к приложению webdemo. Заменим текст страницы access.html на следующий (листинг 16.9).

Листинг 16.9. Страница access.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>Доступ к базе данных</TITLE>
  </HEAD>
  <BODY>
    <H2> Добро пожаловать, <#name>!</H2>
    <P> На этой странице вы можете установить параметры запроса к базе
     данных</P>
    <HR>
    <P><FORM action="<#server>/viewdb" method="POST">
    <P>Категория: <SELECT name="category">
    <OPTION value="Monitors">Monitors</OPTION >
    ...
    </SELECT>
     <P>Цена от: <INPUT type="TEXT" name="minprice" size=5> до: <INPUT
     type="TEXT" name="maxprice" size=5></P>
     <P><INPUT type="SUBMIT" value="Отправить запрос"></P>
     </FORM></P>
     <HR>
```

```
    <P><A href="<#server>/viewdb">Показать всю таблицу</A></P>
  <P><A href="<#server>/logout">Выйти</A></P>
  </BODY>
</HTML>
```

Как видим, теперь страница access.html содержит форму, позволяющую направлять запрос к базе данных, используя значения категории товара и ценового диапазона, а также ссылку, позволяющую получить содержимое таблицы целиком. Форма и ссылка обращаются к новому CGI-сервису `viewdb`, который мы должны создать с помощью редактора свойства `Actions` Web-модуля (назовем новый объект `ViewDB`). Обработчик события `OnAction` для нового сервиса приводится в листинге 16.10.

Листинг 16.10. Обработчик события `OnAction` сервиса `viewdb`

```
void __fastcall TWebModule1::WebModule1ViewDBAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  if (userid == 0)
  {
    Response->StatusCode = 403;
    Response->Content = "Неверный пароль";
    return;
  }
  int ParamCount = 0;
  String ParamString;
  if (Request->ContentFields->Values["category"] != "")
  {
     ParamCount++;
     ParamString = "category = '" +
     Request->ContentFields->Values["category"] + "'";
  }
  if (Request->ContentFields->Values["minprice"] != "")
  {
     ParamCount++;
     if (ParamCount > 1)
     ParamString = ParamString + " and ";
     ParamString = ParamString + "price >= " +
     Request->ContentFields->Values["minprice"];
  }
```

```
  if (Request->ContentFields->Values["maxprice"] != "")
  {
     ParamCount++;
     if (ParamCount > 1)
     ParamString = ParamString + " and ";
     ParamString = ParamString + "price <= " +
     Request->ContentFields->Values["maxprice"];
  }
  String SQLCmd = "select * from pl";
  if (ParamCount > 0)
  SQLCmd = SQLCmd + " where " + ParamString;
  SQLCmd = SQLCmd + " order by category, price";
  IdPriceListDataSet->CommandText = SQLCmd;
  IdPriceListDataSet->Open();
  DataSetTableProducer1->Columns->Items[1]->Title->Caption =
  "Категория";
  DataSetTableProducer1->Columns->Items[2]->Title->Caption =
  "Наименование";
  DataSetTableProducer1->Columns->Items[3]->Title->Caption =
  "Цена";
  DataSetTableProducer1->Columns->Items[4]->Title->Caption =
  "Скидка";
  DataSetTableProducer->Footer->Text =
  "<P><A href='"+Request->ScriptName+"/logout'>Выйти</A></P>";
  Response->Content = DataSetTableProducer1->Content();
  IdPriceListDataSet->Close();
}
```

В этом обработчике мы сначала анализируем параметры поступившего CGI-запроса, и на их основе формируем SQL-запрос к базе данных. Если CGI-запрос не содержит параметров, уточняющих SQL-запрос, формируется SQL-запрос на получение всех записей таблицы. Записи сортируются по значениям полей `category` и `price`.

После этого мы присваиваем строку, содержащую SQL-команду, свойству `CommandText` объекта `IdPriceListDataSet` и открываем набор данных. Теперь можно изменить свойства объектов списка `Columns` объекта `DataSetTableProducer1`. По умолчанию заголовки столбцов HTML-таблицы получают названия, соответствующие названиям полей таблицы базы данных. Мы назначаем заголовкам названия, переведенные на русский язык. Если объекты `THTMLTableColumn` не были созданы нами во время разработки, к ним можно обращаться только после открытия набора данных, так как при

его открытии объект DataSetTableProducer1 получает информацию о столбцах таблицы и динамически создает соответствующие им объекты THTMLTableColumn. Далее мы присваиваем свойству Content объекта Response строку, возвращенную методом Content объекта-генератора. Если бы никаких предварительных операций по настройке объекта DataSetTableProducer1 не требовалось, мы могли бы присвоить ссылку на него свойству Producer объекта ViewDB, представляющего сервис viewdb.

Для форматирования таблицы мы используем обработчик события OnFormatCell (листинг 16.11).

Листинг 16.11. Обработчик события OnFormatCell

```
void __fastcall TWebModule1::DataSetTableProducer1FormatCell(
     TObject *Sender, int CellRow, int CellColumn,
     THTMLBgColor &BgColor, THTMLAlign &Align,
     THTMLVAlign &VAlign, AnsiString &CustomAttrs, AnsiString &CellData)
{
  if ((CellRow << 31) == 0) BgColor = "e8e8e8";
  else BgColor = "ffffff";
}
```

Параметры процедуры-обработчика CellRow и CellColumn указывают строку и столбец, которому принадлежит ячейка (строка с номером 0 — это строка заголовков столбцов, а столбец с номером 0 — самый первый столбец таблицы). Параметр-переменная BgColor позволяет указать фоновый цвет ячейки в принятом в HTML-формате, а параметры-переменные Align и VAlign управляют выравниванием текста ячейки. Параметр-переменная CellData содержит текст ячейки, который мы можем модифицировать.

В нашем обработчике, для того чтобы сделать таблицу простой и удобной для понимания, мы выделяем нечетные строки светло-серым цветом, а четные — белым. Таблица, полученная в результате выполнения запроса к базе данных, показана на рис. 16.7.

Компонент *DataSetPageProducer*

Как и компонент PageProducer, компонент DataSetPageProducer генерирует страницу на основе шаблона, содержащего специальные тэги, которые в процессе генерации контента заменяются HTML-текстом (как и в случае с компонентом PageProducer для этого служит событие OnHTMLTag). Кроме того, компонент DataSetPageProducer позволяет использовать специальные тэги, имена которых соответствуют полям набора данных, с которым компонент DataSetPageProducer связан через свойство DataSet. В результирующем HTML-тексте такие тэги будут заменены значениями одноименных полей текущей записи набора данных.

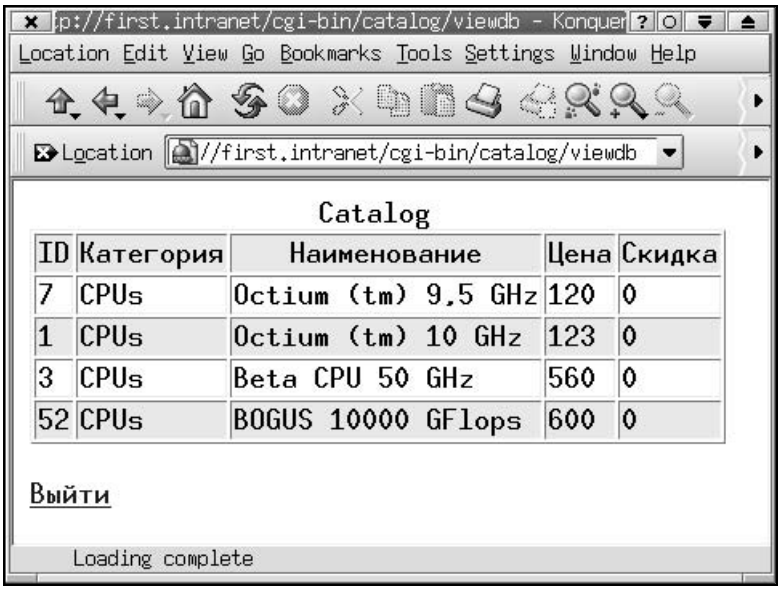


Рис. 16.7. HTML-таблица, содержащая результат запроса к базе данных

Приложения баз данных и технология WebSnap

В этом разделе мы рассмотрим компонент `DataSetAdapter`, расположенный на странице **WebSnap** палитры компонентов.

Компонент `DataSetAdapter`, как и другие компоненты-адаптеры, предоставляет шаблонам страниц доступ к объектам серверного приложения. Как следует из его названия, компонент `DataSetAdapter` позволяет получить доступ к компонентам-наборам данных. Компонент `DataSetAdapter` предоставляет несколько встроенных команд *(подробную информацию о концепции команд адаптера см. в гл. 9)*. Для того, чтобы просмотреть список всех предустановленных команд компонент-адаптера `DataSetAdapter`, необходимо открыть редактор команд (команда **Actions Editor** контекстного меню компонента) и в окне редактора вызвать команду контекстного меню **Add All Actions**. В списке команд адаптера присутствуют команды, позволяющие выполнять практически все операции, которые можно выполнять с наборами данных. Кроме того, будучи связан с открытым набором данных, компонент предоставляет поля, соответствующие полям набора данных. Разумеется, как и другие компоненты-адаптеры, `DataSetAdapter` позволяет программисту создавать собственные команды и поля в дополнение к уже существующим.

В качестве примера использования компонента `DataSetAdapter` напишем приложение, позволяющее просматривать изображения, хранящиеся в базе данных с помощью Web-браузера. Пусть у нас есть таблица базы данных, назовем ее `images`, включающая следующие поля:

- `id` — целочисленный идентификатор записи, первичный ключ;
- `name` — строковое поле, содержащие название изображения;
- `image` — BLOB-поле, содержащее графические данные в одном из форматов, поддерживаемых браузером.

Создадим новое WebSnap-приложение (пункт **WebSnap Application** на странице **WebSnap** диалогового окна **New Items**). В диалоговом окне **New WebSnap Application** в группе **Application Module Components** выберем пункт **Page Module**. Назовем модуль `ImageViewPage`. Откроем окно **Application Module Page Options** (кнопка **Page Options...**) и в группе **Producer** в списке **Type** выберем компонент `AdapterPageProducer`. В результате будет создана форма страничного модуля, содержащая стандартные компоненты WebSnap-приложения. Добавим в эту форму компонент `DataSetAdapter` (рис. 16.8).

Рис. 16.8. Форма страничного модуля `ImageViewPage`

Компонент `DataSetAdapter` нуждается в клиентском наборе данных (`SQLClientDataSet` или `ClientDataSet`). До сих пор в наших простых приложениях-примерах мы размещали компоненты связи с базами данных в том же модуле, что и остальные компоненты приложения. Вообще говоря, Borland рекомендует использовать для этой цели отдельные модули данных, что мы сейчас и сделаем.

Создадим новый модуль данных для WebSnap-приложения (пункт **WebSnap Data Module** страницы **WebSnap** диалогового окна **New Items**). Разместим в форме нового модуля компоненты `SQLConnection` и `SQLClientDataSet`. Объект `SQLConnection1` должен предоставлять доступ к таблице `images`,

объект SQLClientDataSet1 должен быть связан с ним через свойство SQLConnection и содержать стандартный запрос на выборку всех записей из таблицы images:

```
select * from images
```

Внешний вид формы модуля данных показан на рис. 16.9.

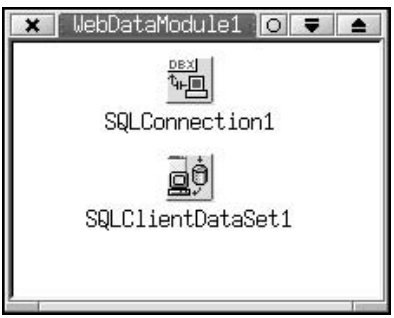


Рис. 16.9. Форма модуля данных

Сохраним все файлы приложения. Для того чтобы клиентский набор данных был доступен объекту DataSetAdapter1, необходимо сделать файл модуля данных видимым в файле страничного модуля. Если мы работаем с языком Delphi Language, внесем файл модуля данных в раздел файла uses страничного модуля. При работе с языком C++ включим заголовочный файл модуля данных в файл страничного модуля с помощью директивы #include.

Теперь свойству DataSet объекта DataSetAdapter1 можно присвоить ссылку на объект SQLDataSet1. Эта ссылка может иметь вид:

```
WebDataModule1->SQLClientDataSet1
```

Убедимся, что однонаправленный набор данных открыт (свойство Active установлено, равным true). Откройте редактор полей объекта DataSetAdapter1 (команда **Fields Editor** контекстного меню компонента-адаптера) и выберем команду контекстного меню **Add All Fields**. После этого в списке полей адаптера должны появиться три автоматически созданных поля: AdaptID, AdaptNAME и AdaptIMAGE, соответствующие полям таблицы images. Обращаем внимание, что для доступа к этим полям адаптера из шаблона нужно использовать имена ID, NAME и IMAGE соответственно.

Получить в шаблоне страницы значения полей ID и NAME для текущей записи можно с помощью конструкции:

```
<%=DataSetAdapter1.Name.Value%>
```

Однако этот способ пригоден только для полей, значения которых можно привести к строковому типу. Для поля `IMAGE` он не подходит.

Для навигации по набору данных можно воспользоваться командами адаптера `FirstRow`, `LastRow`, `NextRow` и `PrevRow`, выполняющих, соответственно, переход к первой, последней, следующей (по отношению к текущей) и предыдущей записям таблицы. Как именно CGI-приложение, не сохраняющее информацию о своем состоянии во время предыдущих вызовов, "узнает", какая запись является следующей или предыдущей, будет сказано ниже.

В сценарии, расположенном в шаблоне страницы, гиперссылку на команду адаптера можно вставить, например, так:

```
<A HREF='<%=DataSetAdapter1.NextRow.AsHREF%>'>Следующая запись</A>
```

Проблему с отображением графического содержимого баз данных в HTML-страницах можно решить с помощью поля адаптера `DataSetAdapterImageField`. В редакторе полей удалим поле `AdaptIMAGE`, откроем окно **Add Web Component** (команда **New Component** контекстного меню или аналогичная кнопка на панели инструментов редактора полей) и в раскрывшемся списке выберем тип нового поля `DataSetAdapterImageField`. В инспекторе объектов назначим свойству `Name` нового поля адаптера значение `ImageField`. В раскрывающемся списке значений свойства `DataSetField` выберем пункт `IMAGE`. Теперь у нас есть поле адаптера `DataSetAdapter`, способное отображать графические данные. Такое поле не могло быть создано автоматически, так как адаптер "не знает" какие именно данные хранятся в поле типа BLOB. Кроме поля `DataSetAdapterImageField` для работы с объектами BLOB, адаптер `DataSetAdapter` предоставляет также поле `DataSetAdapterMemoField`.

Для того чтобы вставить в страницу изображение, полученное из базы данных с помощью поля `DataSetAdapterImageField`, в шаблон страницы следует внести конструкцию:

```
<IMG SRC="<%=DataSetAdapter1.ImageField.Image.AsHREF%>">
```

В результирующей HTML-странице значением параметра `SRC` станет ссылка, по которой клиентская программа сможет загрузить изображение. Хотя у поля данного типа есть обработчик события `OnGetImage`, назначать ему процедуру, как правило, не требуется.

HTML-страница просмотра изображений создаваемого приложения кроме самого изображения будет содержать панель кнопок, вызывающих команды адаптера, которые выполняют навигацию по записям набора данных. Нам не придется вручную писать сценарии в шаблонах страниц, так как их можно сгенерировать автоматически с помощью редактора элементов компонента `AdapterPageProducer`.

Щелкнем кнопкой мыши на пиктограмме объекта AdapterPageProducer в форме страничного модуля. Откроется окно редактора Web-элементов объекта AdapterPageProducer (рис. 16.10)

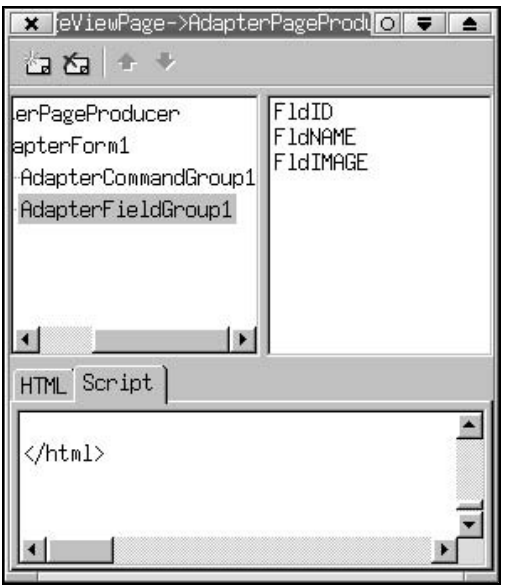


Рис. 16.10. Окно редактора элементов объекта AdapterPageProducer с добавленными элементами

Создадим в редакторе новый элемент AdapterForm. Выделим объект AdapterForm1 и с помощью контекстного меню (команда **New Component**) добавим элемент AdapterFieldGroup. В инспекторе объектов назначим свойству Adapter объекта AdapterFieldGroup1 ссылку на объект DataSetAdapter1. Выделим объект AdapterFieldGroup1 в окне редактора элементов страницы и щелкнем по нему правой кнопкой мыши. В открывшемся контекстном меню выберем пункт **Add All Fields**.

Снова выделим объект AdapterForm1 и с помощью контекстного меню добавим элемент AdapterCommandGroup. В инспекторе объектов назначим свойству DisplayComponent объекта AdapterCommandGroup1 ссылку на объект AdapterFieldGroup1. Выделим в редакторе элементов объекта AdapterPageProducer объект AdapterCommandGroup1 и в контекстном меню выберем команду **Add Commands...** В открывшемся списке выберем команды навигации по набору данных. Текст сгенерированного сценария можно просмотреть в окне вкладки **Script** редактора Web-элементов объекта AdapterPageProducer, а на вкладке HTML можно просмотреть получающуюся в результате выполнения сценария HTML-страницу.

Результат работы программы представлен на рис. 16.11.

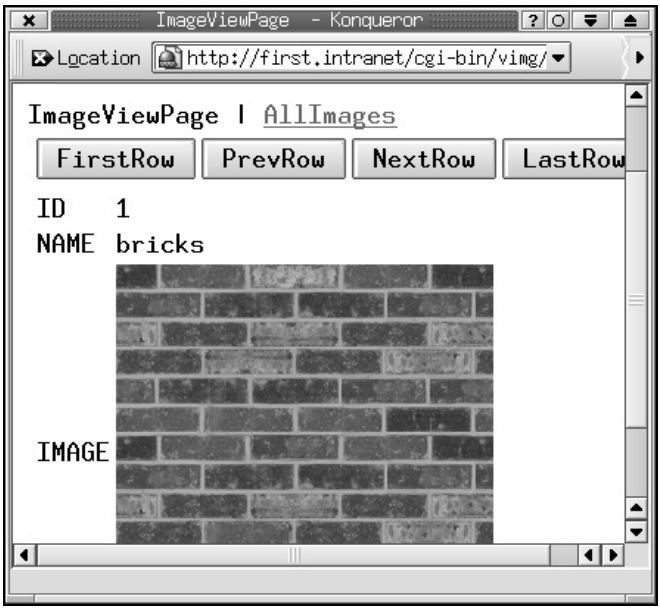


Рис. 16.11. Просмотр изображений, хранящихся в базе данных, с помощью приложения WebSnap

Как же команды, подобные `NextRow` и `PrevRow`, выполняют переход к следующей или предыдущей записи набора данных, ведь для этого нужно знать состояние набора данных во время предыдущего вызова CGI-приложения? Если мы посмотрим внимательно на ссылку, по которой вызывается изображение, то увидим среди параметров CGI-запроса значение поля `ID` соответствующей записи. Значения первичных полей всегда передаются, как часть CGI-запроса, содержащегося в ссылке на команду адаптера. По этой причине навигация с помощью команд адаптера `DataSetAdapter` выполняется успешно только в том случае, если таблица, с которой связан соответствующий набор данных, содержит первичные ключи.

Возможности компонента `DataSetAdapter` не ограничиваются последовательным просмотром содержимого отдельных записей с генерацией новой страницы для каждой записи. При желании мы можем разместить все записи на одной странице так, чтобы они сразу были доступны пользователю. Для размещения всех записей на одной странице в шаблоне страницы следует воспользоваться специальным средством языка сценариев WebSnap — объектом-энумератором (enumerator) и свойством `Records` объекта `DataSetAdapter`. Объекты-энумераторы позволяют перебирать значения элементов наборов и по принципам использования очень похожи на итераторы из стандартной библиотеки шаблонов языка C++.

Листинг 16.12 демонстрирует использование энумераторов для перебора записей набора данных, связанного с объектом DataSetAdapter1.

Листинг 16.12. Перебор записей набора данных с помощью энумератора

```
<% e = new Enumerator(DataSetAdapter1.Records)
for (; !e.atEnd(); e.moveNext())
{ %>
 <P><%=DataSetAdapter1.Name.Value%><BR>
 <IMG SRC="<%=DataSetAdapter1.ImageField.Image.AsHREF%>"></P>
<% }%>
```

Впрочем, если мы хотим показать содержимое всей таблицы базы данных в одной HTML-таблице, это легко можно сделать с помощью того же редактора элементов компонента AdapterPageProducer. В рамках технологии WebBroker мы могли бы воспользоваться компонентом DataSetTableProducer, однако этот компонент не позволяет просматривать изображения. Технология WebSnap решает эту проблему.

В приложении, используемом в качестве примера, создадим новый страничный модуль (пункт **WebSnap Page Module** вкладки **WebSnap** диалогового окна **New Items**), назовем его AllImages, а в качестве компонента-генератора контента выберем компонент AdapterPageProducer. Поместим в форму модуля компонент DataSetAdapter (рис. 16.12).

Рис. 16.12. Форма страничного модуля AllImages

Сделаем модуль данных видимым для компонентов нового страничного модуля AllImages и присвоим объекту DataSetAdapter1 этого модуля ссылку на объект SQLClientDataSet1. Откроем редактор полей объекта DataSetAdapter1

(команда **Fields Editor** контекстного меню компонента-адаптера) и выберем команду контекстного меню **Add All Fields**. Заменим поле `AdaptIMAGE` полем `DataSetAdapterImageField`, как было описано выше. Откроем окно редактора Web-элементов объекта `AdapterPageProducer` и с помощью команды контекстного меню **New Component...** добавим компонент `AdapterForm`. В контекстном меню объекта `AdapterForm1` снова выберем команду **New Component...** и в открывшемся списке — пункт `AdapterGrid`. В инспекторе объектов назначим свойству `Adapter` объекта `AdapterGrid1` ссылку на объект `DataSetAdapter1`. В редакторе элементов объекта `AdapterPageProducer` щелкнем правой кнопкой мыши на строке `AdapterGrid1` и в открывшемся контекстном меню выберем команду **Add All Columns**. Мы создали страницу (рис. 16.13), которая будет отображать содержимое таблицы базы данных (в том числе и изображения).

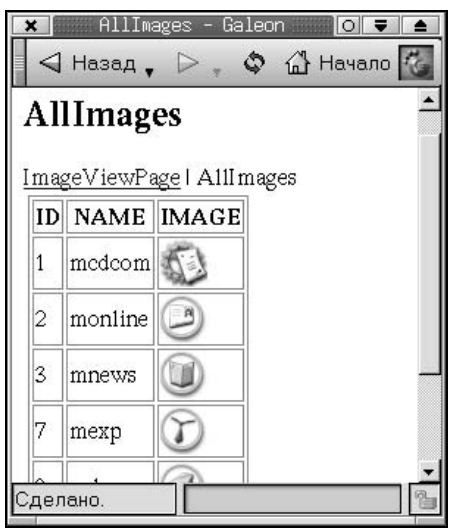


Рис. 16.13. Страница `AllImages` в окне браузера

Часть IV

Профессиональные программные продукты

Глава 17

Создание и распространение пакетов компонентов среды Kylix 3

В этой главе будут рассмотрены вопросы создания и распространения компонентов для среды Kylix 3. Хотя компоненты и пакеты компонентов нельзя назвать законченными программными продуктами, многие программисты и некоторые компании специализируются на разработке пакетов компонентов, распространяя их как бесплатно, так и на коммерческой основе. Даже, если нет цели поставлять пакеты компонентов на рынок, часто имеет смысл объединять в пакеты классы, которые мы планируем активно использовать в разрабатываемых нами приложениях.

Что такое компоненты?

Компоненты — это классы, написанные на языке Delphi Language или C++. От обычных классов компоненты отличаются рядом особенностей. Прежде всего, компоненты регистрируются в палитре компонентов среды Kylix, что позволяет программисту легко вносить их в свои приложения. Компоненты делятся на визуальные и невизуальные. К визуальным относятся в основном компоненты, реализующие элементы графического интерфейса приложения. Невизуальными компонентами являются те, которые во время выполнения программы не предоставляют графических элементов.

Более важной отличительной чертой компонентов является то, что взаимодействовать с ними можно еще на этапе разработки приложения, причем, это взаимодействие является интерактивным. Визуальные компоненты допускают настройку своего внешнего вида, размера и расположения в окне приложения. При этом все вносимые изменения тут же отображаются в редакторе форм (в этом и заключается один из ключевых моментов концепции визуального программирования). С помощью инспектора объектов можно присваивать значения свойствам объектов-компонентов, причем, если разработчик компонента предусмотрел наличие между свойствами ло-

гических связей, эти связи могут быть отражены в инспекторе объектов. Самый простой пример — зависимость множества допустимых значений одного свойства от значения, присвоенного другому свойству.

Но и этим возможности работы с компонентами не ограничиваются. Для свойств сложных типов разработчик компонента может предоставить различные редакторы компонентов и надстройки, облегчающие заполнение этих свойств еще на этапе разработки программы. С редакторами свойств компонентов мы встречались уже не раз при работе со стандартными компонентами Kylix.

На этапе разработки возможно взаимодействие не только со свойствами, но и с отдельными методами компонентов. Прежде всего, это конструкторы и деструкторы, а также метод `Paint` для визуальных компонентов. Кроме того, если операции чтения или записи значений свойств компонента выполняются при помощи его методов, эти методы вызываются при установке значений свойств в инспекторе объектов.

Наиболее яркий пример взаимодействия между компонентами приложения на этапе разработки демонстрируют компоненты, предназначенные для работы с базами данных, позволяющие протестировать работу приложения с БД еще до компиляции самого приложения.

Этапы разработки компонентов

Если мы задумали создать новый компонент, первая задача, которую нам предстоит решить, — выбор класса-прародителя. От того, насколько грамотно мы выберем класс, являющийся основой нашего компонента, будет зависеть дальнейшее удобство использования компонента и его функциональность. В табл. 17.1 перечислены основные базовые классы, используемые при разработке компонентов, и даны рекомендации относительно выбора базового класса в зависимости от функций и назначения разрабатываемого компонента.

Таблица 17.1. *Базовые классы для разработки компонентов*

Класс	Описание
`TComponent`	Этот класс может служить основой для разработки невизуальных компонентов. Он реализует все базовые методы и свойства, необходимые любому компоненту. Основным достоинством данного класса является возможность самостоятельно загружать и сохранять потоки данных в интегрированной среде во время разработки

Таблица 17.1 *(окончание)*

Класс	Описание
`TWidgetControl`	Это базовый класс для создания всех графических компонентов, являющихся окнами (кнопок, панелей, элементов ввода и управления, диалоговых окон и т. п.). Компонент-потомок данного класса должен быть основан на каком-либо классе Qt library. Свойство `Handle`, введенное в классе `TWidgetControl`, должно содержать ссылку на экземпляр соответствующего объекта Qt. В этом случае с данным компонентом можно будет работать стандартными средствами CLXDisplay API
`TGraphicControl`	Используйте этот класс для создания визуального компонента, не являющегося окном. Потомки класса `TGraphicControl` отображаются в клиентской области своих родительских элементов и поэтому требуют значительно меньших системных ресурсов. Класс `TGraphicControl` вводит метод `Paint` и свойство `Canvas`, позволяющие выводить изображение. Вместе с тем этим компонентам присущи определенные ограничения, связанные прежде всего с возможностью реагировать на события системы
`TCustomControl`	Этот класс происходит от класса `TWidgetControl` и добавляет в него такие элементы, как свойство `Canvas` и метод `Paint`. Используйте этот класс для создания компонента, в котором нам необходимо иметь контроль над процессом перерисовки нижележащего графического элемента Qt library, например, добавление дополнительных графических элементов (перерисовка стандартных графических элементов Qt library осуществляется на уровне самой Qt)
Классы семейства `TCustomXXX`	В библиотеке CLX содержится несколько классов, которые можно рассматривать как "заготовки" компонентов, реализующих определенные функции. Как правило, некоторые методы этих компонентов являются абстрактными, а свойства — не публикуются по умолчанию. Такие классы позволяют разработчикам создать несколько "пользовательских" компонентов на основе одного и того же класса и в каждом компоненте публиковать только те из заранее определенных свойств, которые необходимы для данного конкретного компонента. Примерами таких классов-заготовок могут служить `TCustomListBox`, `TCustomDBGrid` и `TcustomPageProducer`

После того, как мы выбрали базовый класс для нашего компонента, мы можем приступить к созданию модуля компонента.

Создание модуля компонента проще всего начать с вызова окна **New Component** (пункт **Component** страницы **New** диалогового окна **New Items**), показанного на рис. 17.1.

Раскрывающийся список **Ancestor Type** служит для выбора класса-прародителя. В строке **Class Name** следует ввести имя класса, реализующего компонент. Список **Palette Page** позволяет выбрать вкладку палитры компонентов, на которой будет размещен новый компонент. Если мы хотим разместить компонент на новой, специально созданной, вкладке палитры, это можно сделать с помощью функции `Register` (*см. ниже*). Элемент ввода **Unit File Name** позволяет указать имя модуля для создаваемого компонента, а строка **Search Path** — отредактировать список каталогов, в которых выполняется поиск элементов компонентов.

После того, как мы нажмем кнопку **OK**, будет создан новый модуль, содержащий заготовку класса нового компонента.

Следующим этапом является определение свойств компонента. Свойства компонентов определяются так же, как и свойства классов, следует только иметь в виду, что в окне инспектора объектов отображаются только свойства, помещенные в разделе `published` класса, реализующего компонент. Обычно для свойств простых типов, предназначенных как для чтения, так и для записи, в разделе декларации класса `private` или `protected` объявляется поле соответствующего типа. По принятой в средствах разработки Borland традиции, имя такого поля совпадает с именем свойства и отличается от него добавлением префикса "F".

Рассмотрим некоторые специальные случаи определения свойств компонентов.

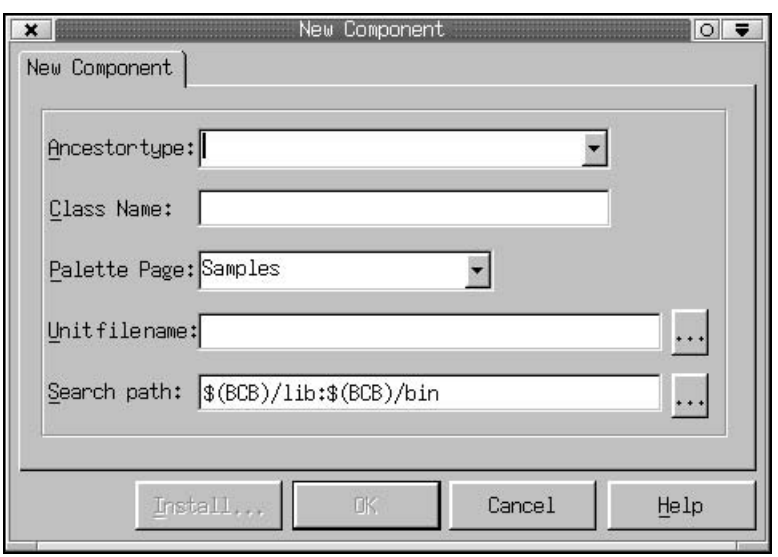


Рис. 17.1. Окно **New Component**

Некоторые свойства позволяют выполнять к ним доступ таким образом, как если бы они представляли собой массивы. Свойства-массивы должны обладать следующими характеристиками:

- свойства-массивы объявляются с использованием одного или нескольких индексных параметров, при этом тип индексов должен быть простым (целое число или строка, но не запись или класс);
- директивы доступа к свойству `read` и `write` должны использовать методы, а не поля компонента;
- если в определении свойства-массива используется несколько индексов (то есть свойство рассматривается как многомерный массив), методы доступа должны содержать параметры для каждого индекса в том же порядке, что и в определении свойства.

Рассмотрим пример. Пусть у нас есть свойство-массив `Strings`, объявленное в компоненте `SomeComponent` как:

```
property Strings[Index: Integer]: String read GetString write PutString;
```

Объявления методов `GetString` и `PutString` должны выглядеть следующим образом:

```
function Get(Index: Integer): String;
procedure Put(Index: Integer; const S: String);
```

В результате к свойству `Strings` компонента `SomeComponent` можно будет обращаться как к массиву элементов типа `String`.

Мы можем объявить свойство-массив таким образом, что оно будет стандартным свойством своего компонента. Это означает, что при использовании компонента программист сможет обращаться с соответствующим объектам так, как если бы они были переменными типа массив.

Например, если объявить свойство `Strings` как:

```
property Strings[Index: Integer]: String read GetString write PutString;
default;
```

объекты, соответствующие компоненту `SomeClass`, можно будет использовать в программе как переменные типа "массив строк".

У класса может быть только одно стандартное свойство-массив, и потомки этого класса не могут переопределять данное свойство.

Свойства компонента сами могут быть объектами или другими компонентами. Например, у компонента `TShape` есть свойства-объекты `TBrush` и `TPen`. Свойство компонента, являющееся объектом, может быть развернуто в окне инспектора объектов таким образом, чтобы можно было получить доступ к его собственным свойствам. Свойства-объекты должны быть потомками класса `TPersistent` для того, чтобы их публикуемые свойства (то есть свойства, объявленные в разделе `published`) могли быть записаны в поток данных и отображены в окне инспектора объектов.

Обычно свойству-объекту соответствует поле компонента, представляющее собой ссылку на класс объекта. В конструкторе компонента следует создать экземпляр объекта и присвоить ссылку на него соответствующему полю.

Если свойство-объект не предназначено только для чтения, механизм присваивания нового значения этому свойству следует организовать с помощью метода, использующего метод `Assign` для присвоения нового значения (напомним, что в свойствах можно использовать классы-потомки `TPersistent`, которые наследуют от своего прародителя метод `Assign`). Пусть, например, в компоненте `SomeComponent` определено свойство:

```
property SomeObject : TSomeClass read FSomeObject write SetSomeObject;
```

Метод `SetSomeObject` может выглядеть, как в листинге 17.1.

Листинг 17.1. Схема метода установки значения свойству-объкту

```
procedure TSomeComponent.SetSomeObject(Value: TSomeClass);
begin
  if Assigned(Value) then
  FSomeObject.Assign(Value);
end;
```

В деструкторе компонента свойства-объекты следует уничтожать до вызова родительского деструктора (листинг 17.2).

Листинг 17.2. Деструктор компонента

```
procedure TSomeComponent.Destroy;
begin
  FSomeObject.Free;
  inherited Destroy;
end;
```

События компонентов представляют собой свойства процедурного типа, соответствующего процедуре-обработчику. Поскольку обработчики событий являются методами объектов, объявления соответствующих типов должны содержать особые директивы. Например, для языка Delphi Language:

```
type
  TSomeEvent = procedure(Sender: TObject) of object;
```

или для языка C++:

```
typedef void __fastcall (__closure *TSomeEvent)(System::TObject *
Sender);
```

Свойства-события используют поля соответствующего процедурного типа. Свойства-события должны размещаться в разделе `published`. Как же инспектор объектов различает обычные свойства и свойства-события? Очень

просто. На вкладку **Events** помещаются все свойства процедурного типа, чьи названия имеют префикс **"On"**.

Добавление в компонент методов не отличается от их добавления в любой другой объект. Тем не менее, существует несколько моментов, которые следует учитывать при разработке компонентов.

Одна из главных целей, преследуемых при создании компонентов, — упростить использование соответствующих классов для программиста. Поэтому следует по возможности исключить взаимозависимость методов. Интерфейс компонента должен быть спроектирован таким образом, чтобы результаты вызова одного метода не зависели от результатов вызовов других методов (разумеется, это правило не является абсолютным).

Создавая новый компонент, мы имеем возможность переопределить конструктор класса-прародителя и определить собственный. При этом необходимо соблюдать некоторые меры предосторожности. Объявляя конструктор при построении потомка класса `TComponent`, всегда включаем директиву `override`.

Несмотря на то, что с точки зрения синтаксиса директива `override` не является обязательной, ее отсутствие может вызвать проблемы при использовании нашего компонента. Дело в том, что в процессе использования компонента (как во время разработки, так и во время выполнения) невиртуальный конструктор не будет вызван кодом, создающим этот компонент посредством ссылки на класс (например, в потокообразующей системе).

Не следует забывать также добавить вызов унаследованного конструктора в код конструктора нашего компонента (листинг 17.3).

Листинг 17.3. Конструктор компонента

```
constructor TSomeComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
end;
```

Взаимодействие между компонентами и средой разработки

Для того чтобы избежать многих распространенных ошибок, возникающих в процессе создания компонентов, необходимо иметь в виду, что механизм взаимодействия между средой разработки и компонентом во время написания программы максимально приближен к механизму взаимодействия между компонентом и программой во время выполнения. Этот подход является частью более общей концепции "компиляции в процессе разработки", реализованной в средствах визуальной разработки.

Примечание

В настоящее время данная концепция используется в продуктах Borland (и не только) чрезвычайно широко. Именно компиляция в процессе разработки позволяет реализовать в редакторе текста программы такие удобные функции, как автоматическое завершение строки.

Также как программа, прежде чем обращаться к свойствам и методам объекта некоторого класса, должна создать экземпляр объекта, вызвав конструктор соответствующего класса, среда разработки должна вызвать конструктор компонента при помещении его в форму. Только после этого становится возможной работа с визуальными компонентами в форме приложения и работа со свойствами компонентов в инспекторе объектов. При размещении визуального компонента в форме приложения, а также при изменении его размеров, среда разработки, кроме прочего, вызывает методы компонента, ответственные за отрисовку его графических элементов. Обычно с вызовами этих методов, а также других методов компонента, к которым обращается среда разработки, не возникает проблем. Однако иногда разработчику компонента может потребоваться, чтобы поведение компонента во время разработки отличалось от его поведения во время выполнения программы. Кстати, можно потребовать отмены выполнения некоторых действий, когда компонент находится в состоянии разработки.

Методы компонента могут определить, в каком состоянии находится компонент при помощи свойства `ComponentState`. Данное свойство представляет собой набор (set), в котором может присутствовать ряд значений (флагов), характеризующих состояние компонента. В табл. 17.2 перечислены значения, которые могут входить в набор свойства `ComponentState`, и описаны соответствующие им состояния компонента.

Таблица 17.2. *Элементы набора ComponentState и соответствующие им состояния компонента*

Элемент	Состояние компонента
`csDesigning`	Наличие этого элемента в наборе свойства `ComponentState` означает, что объект, соответствующий компоненту, находится в режиме разработки (в форме редактора форм)
`csDesignInstance`	Флаг устанавливается для компонента, являющегося корневым компонентом редактора
`csAncestor`	Флаг устанавливается, если компонент был введен в форме, являющейся прародителем данной формы. Он может быть установлен только, если установлен флаг `csDesigning`
`csDestroying`	Флаг устанавливается, если объект уничтожается
`csFixups`	Флаг устанавливается, если данный компонент связан с компонентом другой формы, которая еще не загружена. Он сбрасывается после того, как все подобные связи будут разрешены

Таблица 17.2 *(окончание)*

Элемент	Состояние компонента
`csFreeNotification`	Этот флаг означает, что другие компоненты требуют, чтобы данный компонент оповещал их о своем уничтожении. Устанавливается при вызове метода `FreeNotification` данного компонента
`csInline`	Флаг устанавливается для компонента, являющегося частью формы и содержащего другие компоненты
`csLoading`	Загрузка объекта компонента из файлового потока. Этот флаг устанавливается сразу после создания компонента и сбрасывается только тогда, когда все дочерние элементы компонента полностью загружены
`csReading`	Считывание значений свойств из потока
`csUpdating`	Компонент обновляется с целью отображения изменений, выполненных в форме-прародителе. Устанавливается только при установленном флаге `csAncestor`
`csWriting`	Запись свойств компонента в поток

При разработке компонентов чаще всего проверяется состояние флага `csDesigning`, позволяющее определить, находится ли компонент в режиме разработки. Проверить состояние флага можно с помощью следующих конструкций на языке Delphi Language:

```
if csDesigning in ComponentState then ... else ...;
```

и на языке C++:

```
if (ComponentState.Contains(csDesigning)) ...; else ...;
```

Следует отметить, что флаг `csDesigning` не будет установлен до тех пор, пока конструктор компонента не вызовет унаследованный конструктор. Эти условия почти всегда выполняются при работе с компонентом в редакторе форм.

Регистрация компонента

В процессе регистрации компонент помещается на одну из вкладок палитры компонентов Kylix. Если для создания модуля компонента использовалось окно **New Component**, среда разработки уже сгенерировала код, необходимый для регистрации нового компонента. Если же мы создаем компонент вручную, в модуль созданного компонента следует включить процедуру `Register`.

Процедура `Register` вызывает процедуру `RegisterComponents` для каждого регистрируемого компонента. Процедуре `RegisterComponents` передаются

два параметра: имя вкладки, в которой будет размещаться компонент, и массив, элементами которого являются имена классов компонентов. Пример использования процедуры `Register` показан в листинге 17.4.

Листинг 17.4. Использование процедуры `Register`

```
Unit SomeUnit;
interface
  TSomeComponent = class(TComponent)
  ...
  end;
  procedure Register;
implementation
{$R *.dcr}
...
  procedure Register;
  begin
    RegisterComponents('SomePage',  [TSomeComponent,
    TSomeOtherComponent]);
  end;
end.
```

Вариант процедуры `Register` для языка C++ представлен в листинге 17.5.

Листинг 17.5. Процедура `Register` для языка C++

```
void __fastcall PACKAGE Register()
{
  TComponentClass classes[1] = {__classid(TSomeComponent)};
  RegisterComponents("SomePage", classes, 0);
}
```

Здесь `SomePage` — название вкладки палитры компонентов, на которой будет размещен компонент `SomeComponent`. Если вкладки с таким именем не существует, она будет создана.

Обратим также внимание на директиву `{$R *.dcr}`. Эта директива указывает компилятору на необходимость включения в модуль компонента файла с расширением .dcr и именем, совпадающим с именем модуля. Этот файл содержит ресурсы компонента, в частности, его пиктограмму. К сожалению, в состав пакета Kylix не входит приложение-редактор файлов ресурсов, позволяющее создавать dcr-файлы, так что, если мы хотим, чтобы пиктограм-

ма нашего компонента отличалась от той, которую палитра компонентов назначит ему по умолчанию, нам придется воспользоваться редактором Image Editor, входящим в состав среды Delphi для ОС Windows.

Пакеты компонентов

Пакеты компонентов представляют собой совокупность модулей, реализующих компоненты, объединенные некоторым общим признаком. Не следует рассматривать пакеты просто как хранилища компонентов. Использование пакетов облегчает выполнение таких задач, как регистрация компонентов, создание дополнительных средств работы с ними на этапе разработки приложения и распространение приложений, использующих одни и те же компоненты.

Пакеты Kylix представляют собой разделяемые библиотеки (so-файлы). Обычно имя библиотеки, содержащей пакет, начинается с префикса "bpl".

Пакеты времени разработки и выполнения

Пакеты, с которыми мы будем иметь дело как программисты или как разработчики пакетов, можно разделить на три категории на основании того, на каком этапе работы с программой эти пакеты используются. Эти три категории пакетов перечислены ниже.

1. Пакеты времени выполнения. Пакеты этого типа содержат код, компоненты и другие ресурсы, необходимые приложению во время выполнения. Если приложение использует компоненты из пакета времени выполнения, в случае отсутствия этого пакета приложение работать не будет. Если мы распространяем приложение вместе с пакетами времени выполнения, другие программисты не смогут использовать компоненты из этих пакетов (по крайней мере, традиционным способом).
2. Пакеты времени разработки. Эти пакеты содержат компоненты, редакторы свойств или компонентов, эксперты и все прочие элементы, необходимые для взаимодействия с компонентами на этапе создания приложения в интегрированной среде. Пакеты этого типа используются только в среде разработке и никогда не распространяются вместе с создаваемыми приложениями.
3. Пакеты времени разработки и времени выполнения. Эти пакеты объединяют в себе возможности пакетов первых двух типов и обычно используются в случае отсутствия специальных элементов для работы с компонентами на этапе разработки (таких как редакторы свойств и компонентов или эксперты). Пакеты этого типа можно создавать для упрощения процесса разработки и развертывания использующих их приложений. Однако, если пакет времени разработки и выполнения все

же содержит элементы, необходимые только во время разработки, использование такого пакета в приложениях приведет к неоправданному увеличению объема, занимаемого приложением и его элементами на диске, так как все ненужные во время выполнения программы элементы все равно будут включены в нее.

Пакеты времени выполнения предоставляют разработчику программы возможность выбора: включать ли код пакета в приложение или распространять его отдельно в виде разделяемой библиотеки. Преимуществом второго варианта является возможность разделения кода между приложениями, использующими один и тот же пакет, в результате чего общий объем дискового пространства, необходимый набору программ, уменьшается. Однако, по нашему мнению, в среде Linux распространять пакеты времени выполнения отдельно от приложений не имеет смысла. Реальная экономия дискового пространства может быть достигнута только в том случае, если у конечного пользователя установлено несколько Kylix-приложений, использующих данный пакет, причем, все эти приложения используют пакет как разделяемую библиотеку и придерживаются одного соглашения относительно места размещения ее в файловой системе. На практике распространение пакетов времени выполнения в виде разделяемых библиотек лишь создает дополнительные сложности.

Установить режим использования пакетов времени выполнения приложением можно с помощью флажка **Build with runtime packages** диалогового окна **Project Options** (команда меню **Component | Install Packages...**). По умолчанию код необходимых приложению пакетов включается в исходный код приложения. Для того чтобы использовать пакеты как разделяемые библиотеки, установим флажок **Build with runtime packages**. После этого в строке ввода мы сможем отредактировать список пакетов, с которыми приложение должно быть связано динамически.

Создание пакета компонентов

Создать новый пакет можно, выбрав пункт **Package** на странице **New** диалогового окна **New Items**. После этого будет открыто окно менеджера пакета, напоминающее окно менеджера проекта Kylix (рис. 17.2).

Кнопка **Options** позволяет указать различные параметры пакета, в том числе его тип, а также параметры имени двоичного файла пакета — префикс, расширение, номер версии. С помощью кнопок **Add** и **Remove** можно добавлять модули компонентов в пакете или удалять их из него. Кнопка **Compile** позволяет скомпилировать пакет. Компиляция пакета выполняется так же, как и компиляция приложения (в процессе компиляции могут выдаваться предупреждающие сообщения или сообщения об ошибках). С помощью кнопки **Install** можно зарегистрировать пакет и его компоненты в середе разработки.

Рис. 17.2. Окно менеджера пакета

В папке **Contains** указываются модули компонентов, входящих в новый пакет. Существует несколько правил, которые следует соблюдать при помещении модулей в папку **Contains**:

- модуль не должен содержаться в разделе `contains` другого пакета или в разделе `uses` модуля другого пакета;
- модули, перечисленные в разделе `contains` пакета прямо или косвенно (в разделе `uses` модулей пакета), не могут быть в указаны в папке **Requires** данного пакета (в этом нет необходимости, ведь эти модули и так будут присутствовать в скомпилированном пакете);
- нельзя помещать имя модуля в папку **Contains** пакета, если оно уже находится в разделе `contains` другого пакета, используемого тем же самым приложением.

В папке **Requires** указываются пакеты, содержащие классы, необходимые данному пакету. Некоторые пакеты добавляются в эту папку по умолчанию. Если создаваемый нами пакет использует классы из пакетов, не указанных по умолчанию, мы можем добавить их сами с помощью кнопки **Add**. Если мы создаем два пакета одних и тех же компонентов, один — времени разработки, другой — времени выполнения, разместим все пользовательские компоненты в пакете времени выполнения, а затем создадим пакет времени разработки, добавим в него модули, необходимые для работы с компонентами во время разработки, и укажем пакет времени выполнения в папке **Requires** нового пакета времени разработки.

При внесении пакетов в папку **Requires** рекомендуется придерживаться следующих правил:

- избегать циклических ссылок: пакет `Package1` не может иметь в своей папке **Requires** ссылку на самого себя или на другой пакет, содержащий пакет `Package1` в своей папке **Requires**;
- цепочка ссылок на пакеты не должна содержать рекурсивных ссылок.

Использование пакетов позволяет не только отделить код, необходимый во время разработки от остального кода компонентов, но и разместить функции регистрации компонентов в палитре компонентов в отдельном модуле.

Пакеты, разделяемые библиотеки и директива *$WeakPackageUnit*

Директива `$WeakPackageUnit` предназначена для использования в модулях, ссылающихся на разделяемые библиотеки, которые могут быть не установлены в системе. С помощью этой директивы пакеты, содержащие такие модули, могут регистрироваться в среде разработки и использоваться в программах, не вызывая исключительных ситуаций, даже если соответствующая библиотека отсутствует в системе (разумеется, мы не сможем использовать только те классы пакета, которые не нуждаются в отсутствующей библиотеке). На наш взгляд, в пакетах для Kylix лучше отказаться от директивы `$WeakPackageUnit` и использовать динамическую загрузку разделяемых библиотек *(см. гл. 1)*. Дело в том, что в ОС Linux разработчик компонентов (и приложений) может столкнуться не только с ситуацией, когда нужная библиотека не установлена в системе, но и ситуацией, когда в одном дистрибутиве данная библиотека располагается в одном каталоге, а в другом дистрибутиве — в другом. Лучше всего предусмотреть возможность динамической загрузки разделяемых библиотек в секции `initialization` модуля компонента. В конструктор можно добавить код, проверяющий, была ли необходимая разделяемая библиотека загружена динамически в процессе инициализации модуля или нет. Если библиотека не была загружена, конструктор может вызвать исключительную ситуацию с поясняющим сообщением. Подробнее о взаимодействии приложений с разделяемыми библиотеками будет сказано в следующей главе.

Глава 18

Распространение и настройка Kylix-приложений

В этой главе мы рассмотрим несколько вопросов, связанных с распространением профессиональных приложений, созданных с помощью среды Kylix. Речь пойдет о создании справочной системы Kylix-приложения, распространении приложений в форме исходных текстов и создании дистрибутивов готовых приложений. Мы также коснемся вопросов настройки приложений для работы в ОС Linux.

Создание справочной системы для Kylix-приложения

В ОС Linux отсутствует стандарт справочной системы для графических приложений. Это объясняется тем, что графическая среда не является частью ОС Linux и большинство дистрибутивов предоставляет целый набор графических оболочек. Интерфейсы программирования некоторых из этих оболочек предоставляют готовые средства для разработки справочной системы, однако в случае Kylix нам придется самим позаботиться об этом. Механизм подключения справочных средств к приложению Kylix организован таким образом, чтобы приложение могло использовать самые разные средства просмотра контекстной справки, в том числе и специализированные приложения, созданные сторонними производителями. Общая схема взаимодействия Kylix-приложения со справочной системой приводится на рис. 18.1.

В Kylix-приложении взаимодействие со встроенной справочной системой обеспечивается при помощи глобального объекта `HelpSystem`. Этот объект создается при запуске приложения и является посредником между программным кодом (объекты классов `TApplication`, `TForm` и других классов-потомков `TControl`) и средствами просмотра справки конкретной справочной системы. Приложения направляют запрос на получение справки объекту `HelpSystem`, который пересылает эти запросы модулям, реализующим взаимодействие со справочной системой, созданной разработчиком программы или полученной от стороннего производителя. Вывод запрошенной

справочной информации на экран осуществляется самой справочной системой и никак не контролируется объектом `HelpSystem`.

Элементы управления позволяют генерировать запросы к справочной системе в ответ на стандартные действия пользователя, такие как нажатие клавиши <F1>. Для вызова справочной системы можно также воспользоваться методом `InvokeHelp`, определенном в классе `TApplication` и всех классах-наследниках `THandleComponent` и `TWidgetControl`.

Класс `TControl`, являющийся предком всех элементов управления, предоставляет ряд свойств, позволяющих указать имя файла справки, который содержит встроенную справку этого элемента управления (свойство `HelpFile`), а также идентификатор темы или строку с ключевым словом, идентифицирующим раздел внутри этого файла (свойства `HelpContext` и `HelpKeyword`). Свойство `HelpContext` существует для совместимости с Windows-системой. В Kylix следует использовать справку, основанную на ключевых словах. Выбрать тип справочной системы можно с помощью свойства `HelpType`.

После того, как взаимодействие со справочной системой налажено, все что необходимо сделать, — это установить указанные свойства для каждого элемента управления при разработке программы. Если мы не определили файл справки для конкретного элемента управления, Kylix будет использовать справочный файл предка формы. Если же файл справки не определен и для формы, будет использован файл справки приложения (свойство `Application.HelpFile`). При нажатии пользователем на клавишу <F1> в ходе выполнения программы запрос на получение справочной информации будет перенаправлен через объект `HelpSystem` к `HelpManager` и затем — средству просмотра справки.

Замечание

Обратим внимание на то, что хотя модуль просмотра справки входит в состав приложения, программный код приложения не обращается к нему напрямую, а обращение идет через объект `HelpManager`.

Эта архитектура позволяет облегчить переход приложения от одной справочной системы к другой, а также позволяет одной программе использовать несколько разных справочных систем.

Структура справочной системы для Kylix-приложения выглядит следующим образом: справочная система состоит из *разделов*. Каждый раздел может быть связан с одним или несколькими *ключевыми словами* (с одним ключевым словом может быть связано также несколько разделов). При вызове справки объект `HelpSystem` сначала проверяет, существует ли для данного ключевого слова раздел справочной системы. Если для данного ключевого слова существует один раздел, объект `HelpSystem` посылает справочной системе запрос на вывод соответствующего раздела. Если данное ключевое слово связано с несколькими разделами, объект `HelpSystem` запрашивает список разделов и выводит диалоговое окно, позволяющее выбрать раздел из списка. После этого дается команда на отображение соответствующего раздела.

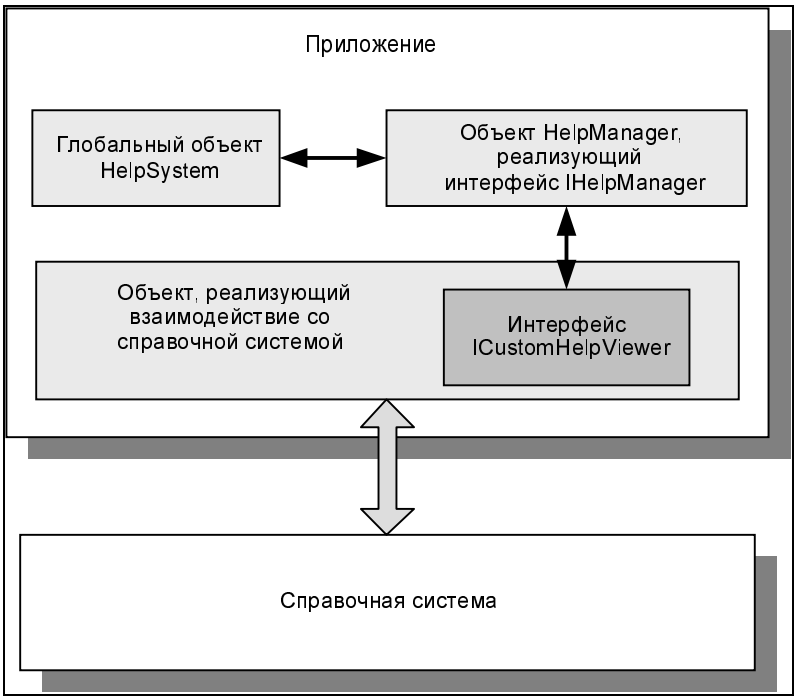


Рис. 18.1. Взаимодействие Kylix-приложения со справочной системой

Для разработчика программы процесс подключения модуля просмотра справочной системы к приложению сводится к созданию класса, реализующего интерфейс `ICustomHelpViewer` или `IExtendedHelpViewer` и осуществляющего взаимодействие со средством просмотра справки.

В табл. 18.1 описываются методы интерфейса `ICustomHelpViewer`, вызываемые приложением для вывода справочной информации.

Таблица 18.1. *Методы интерфейса* `ICustomHelpViewer`

Метод	Описание
`GetViewerName`	`HelpManager` вызывает эту функцию для получения строки, содержащей имя модуля просмотра. Это имя является идентификатором средства просмотра в приложении
`UnderstandsKeyword`	Эта функция вызывается для определения числа разделов, связанных с данным ключевым словом. Если функция возвращает значение 0, значит, ключевое слово справочной системой не распознано

Таблица 18.1 *(окончание)*

Метод	Описание
`GetHelpStrings`	Эта функция используется для получения списка имен разделов, связанных с данным ключевым словом. Вызывается, только если функция `UnderstandsKeyword` вернула значение, отличное от нуля. Список имен разделов возвращается в объекте `TstringList`
`CanShowTableOfContents`	С помощью этого метода приложение проверяет, может ли справочная система выводить содержание (список всех имеющихся разделов)
`ShowTableOfContents`	В ответ на вызов этого метода справочная система должна отобразить содержание (этот метод вызывается только, если вызов `CanShowTableOfContents` вернул значение `true`)
`ShowHelp`	В ответ на вызов этого метода справочная система должна отобразить раздел справки, имя которого передано в качестве параметра метода
`NotifyID`	С помощью этого метода объекту, реализующему интерфейс `ICustomHelpViewer`, передается уникальный числовой идентификатор, который затем используется при высвобождении ресурсов
`SoftShutDown`	Этот метод вызывается для выключения всех внешних подсистем без выключения модуля просмотра. Модуль просмотра может закрыть любое окно, созданное им, и выключить любую внешнюю подсистему
`ShutDown`	Этот метод вызывается для полного выключения модуля просмотра. Модуль просмотра должен выключить все внешние подсистемы и освободить все выделенные ему ресурсы

Как правило, объект, обеспечивающий связь с системой просмотра справки, реализуется в виде отдельного модуля и подключается к основному модулю приложения в операторе `uses` или с помощью менеджера проектов. В процессе запуска приложения соответствующий объект, предоставляющий интерфейс `ICustomHelpViewer`, должен быть создан и зарегистрирован. В процессе завершения программы должно быть выполнено уничтожение этого объекта.

В качестве примера напишем средство просмотра справки для приложения Delphi Language. Справочным файлом будет обычный HTML-файл, а средством просмотра — компонент `TextBrowser`. Хотя, как говорилось выше,

средство просмотра справки может быть независимым приложением, в нашем случае справочная система будет частью самой программы, для которой она используется.

В рассмотренном ниже примере справочный HTML-файл имеет следующую структуру: раздел справочной системы отмечается тэгом-привязкой (anchor) `<A NAME=...>`. Каждому ключевому слову соответствует только один раздел и имя раздела совпадает с ключевым словом. Наша справочная система может отображать содержание, для этого в ней предусмотрен специальный раздел с именем `toc`.

В листинге 18.1 приводится текст простейшего HTML-файла для нашей справочной системы.

Листинг 18.1. HTML-файл для справочной системы Kylix-приложения

```
<HTML>
<HEAD><TITLE></TITLE></HEAD>
<BODY>
<A name=toc>
<H2>Содержание</H2>
<P>
<A href=#okbutton>Кнопка OK</A><BR>
<A href=#cancelbutton">Кнопка Cancel</A>
</P>
<HR>
<A name=okbutton></A>
<H3>Кнопка OK</H3>
<P>
<IMG src="kylix.gif"> Кнопка <B>OK</B> закрывает диалоговое окно
и подтверждает внесенные изменения
</P>
<HR>
<A name=cancelbutton></A>
<H3>Кнопка Cancel</H3>
<P>
<IMG src="kylix.gif"> Кнопка <B>Cancel</B> закрывает диалоговое
окно без сохранения изменений
</P>
</BODY>
</HTML>
```

Из листинга видно, что наш файл справочной системы содержит три раздела: `toc` (содержание), `okbutton` и `cancelbutton`, содержащие информацию о кнопках **OK** и **Cancel** некоторой формы.

Наша следующая задача — создать форму для просмотра справки. Форма будет содержать компонент `TextBrowser` и три кнопки: **Вперед** (`ForwardButton`), **Назад** (`BackwardButton`) и **Содержание** (`TocButton`). С помощью этих кнопок пользователь справочной системы сможет выполнять навигацию по сложному справочному файлу, содержащему перекрестные ссылки между разделами. В модуле этой формы мы сможем разместить описание класса, реализующего интерфейс `ICustomHelpViewer` и объект этого класса. Исходный текст модуля формы приводится в листинге 18.2.

Листинг 18.2. Модуль справочной системы

```
unit HTMLHelp;

interface

uses
  SysUtils, Types, Classes, HelpIntfs, Variants, QGraphics, QControls,
  QForms, QDialogs, QComCtrls, QStdCtrls, QExtCtrls;

type
  THelpViewerForm = class(TForm)
    Panel1: TPanel;
    ForwardButton: TButton;
    BackwardButton: TButton;
    TOCButton: TButton;
    TextBrowser1: TTextBrowser;
    procedure FormShow(Sender: TObject);
    procedure ForwardButtonClick(Sender: TObject);
    procedure BackwardButtonClick(Sender: TObject);
    procedure TOCButtonClick(Sender: TObject);
    procedure TextBrowser1HighlightText(Sender: TObject;
      const HighlightedText: WideString);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  HelpViewerForm: THelpViewerForm;

implementation

{$R *.xfm}

const
  VIEWER_NAME = 'HTML Help Viewer';

type
  THTMLHelpViewer = class (TInterfacedObject, ICustomHelpViewer)
  private
    FHelpManager : IHelpManager;
    FViewerID : Integer;
    FormCreated : Boolean;
    procedure __ShutDown;
  procedure __ShowViewerForm;
  public
    constructor Create;
    function GetViewerName : String;
    function UnderstandsKeyword(const HelpString : String) : Integer;
    function GetHelpStrings(const HelpString : String) : TStringList;
    function CanShowTableOfContents : Boolean;
    procedure ShowTableOfContents;
    procedure ShowHelp(const HelpString: String);
    procedure NotifyID(const ViewerID : Integer);
    procedure SoftShutDown;
    procedure ShutDown;
  end;

var
  HelpViewer : THTMLHelpViewer;

  constructor THTMLHelpViewer.Create;
  begin
    inherited Create;
    FormCreated := False;
  end;
```

```
procedure THTMLHelpViewer.__ShutDown;
begin
  if Assigned(FHelpManager) then
  FHelpManager.Release(FViewerID);
  ShutDown;
end;

procedure THTMLHelpViewer.__ShowViewerForm;
var
  HF : String;
begin
  if not Assigned(HelpViewerForm) then
  begin
    HelpViewerForm := THelpViewerForm.Create(nil);
    FormCreated := True;
  end;
  HF := FHelpManager.GetHelpFile;
  if HelpViewerForm.TextBrowser1.FileName <> HF then
  HelpViewerForm.TextBrowser1.FileName := HF;
  HelpViewerForm.ForwardButton.Enabled :=
  HelpViewerForm.TextBrowser1.CanGoForward;
  HelpViewerForm.BackwardButton.Enabled :=
  HelpViewerForm.TextBrowser1.CanGoBackward;
  HelpViewerForm.Show;
end;

function THTMLHelpViewer.GetViewerName;
begin
  Result := VIEWER_NAME;
end;

function THTMLHelpViewer.UnderstandsKeyword;
begin
  Result := 1;
end;

function THTMLHelpViewer.GetHelpStrings;
```

```
begin
  Result := TStringList.Create;
  Result.Add(HelpString);
end;

function THTMLHelpViewer.CanShowTableOfContents;
begin
  Result := True;
end;

procedure THTMLHelpViewer.ShowTableOfContents;
begin
  __ShowViewerForm;;
  HelpViewerForm.TextBrowser1.ScrollToAnchor('toc');
end;

procedure THTMLHelpViewer.ShowHelp;
begin
  __ShowViewerForm;
  HelpViewerForm.TextBrowser1.ScrollToAnchor(HelpString);
end;

procedure THTMLHelpViewer.NotifyID;
begin
  FViewerID := ViewerID;
end;

procedure THTMLHelpViewer.SoftShutDown;
begin
  if FormCreated then
  begin
    HelpViewerForm.Close;
    HelpViewerForm.Free;
  end;
end;

procedure THTMLHelpViewer.ShutDown;
```

```
  begin
    SoftShutDown;
    FHelpManager := nil;
  end;

procedure THelpViewerForm.FormShow(Sender: TObject);
begin
  Caption := 'Справочная система — ' + Application.Title;
end;

procedure THelpViewerForm.ForwardButtonClick(Sender: TObject);
begin
  TextBrowser1.Forward();
  ForwardButton.Enabled := TextBrowser1.CanGoForward;
  BackwardButton.Enabled := TextBrowser1.CanGoBackward;
end;

procedure THelpViewerForm.BackwardButtonClick(Sender: TObject);
begin
  TextBrowser1.Backward();
  ForwardButton.Enabled := TextBrowser1.CanGoForward;
  BackwardButton.Enabled := TextBrowser1.CanGoBackward;
end;

procedure THelpViewerForm.TOCButtonClick(Sender: TObject);
begin
  TextBrowser1.ScrollToAnchor('toc');
  ForwardButton.Enabled := TextBrowser1.CanGoForward;
  BackwardButton.Enabled := TextBrowser1.CanGoBackward;
end;

procedure THelpViewerForm.TextBrowser1HighlightText(Sender: TObject;
  const HighlightedText: WideString);
begin
  ForwardButton.Enabled := TextBrowser1.CanGoForward;
  BackwardButton.Enabled := TextBrowser1.CanGoBackward;
end;
```

```
initialization
  if not Assigned(HelpViewer) then
  begin
    HelpViewer := THTMLHelpViewer.Create;
    HelpIntfs.RegisterViewer(HelpViewer, HelpViewer.FHelpManager);
  end;

finalization
  if Assigned(HelpViewer) then
  begin
    HelpViewer.__ShutDown;
  end;
end.
```

Роль посредника между объектом `HelpSystem` приложения и объектом `HelpViewerForm` класса `THelpViewerForm` выполняет объект `HelpViewer` класса `THTMLHelpViewer`.

Замечание

Обратим внимание на то, что и класс `THTMLHelpViewer`, и соответствующий ему объект объявляются в разделе `implementation` модуля.

Это оправдано, так как другие модули приложения не будут обращаться к объекту `HelpViewer` непосредственно. В принципе, объявление класса `THelpViewerForm` и объект `HelpViewerForm` также можно было бы разместить в разделе `implementation`, поскольку единственным объектом в приложении, который будет обращаться к объекту `HelpViewerForm`, является объект `HelpViewer` (объект `HelpViewerForm` играет роль средства просмотра справочной системы, которое, напомним, могло бы быть отдельным приложением). Однако раз уж форма `HelpViewerForm` является частью приложения, мы помещаем ее в общедоступный раздел модуля `HTMLHelp`, чтобы приложение могло создать и зарегистрировать эту форму обычным порядком.

Как же объект `HelpSystem` "узнает" об объекте `HelpViewer`? В разделе `initialization` мы создаем экземпляр объекта класса `THTMLHelpViewer` и регистрируем его с помощью процедуры `RegisterViewer`, объявленной в модуле `HelpIntfs`. При этом регистрируемый объект получает ссылку на интерфейс `IHelpManager`, реализуемый объектом `HelpManager`.

В классе `THTMLHelpViewer` объявлены два метода, видимые только внутри модуля `HTMLHelp`. Это метод `__ShutDown`, вызываемый в разделе `finalization`, и метод `__ShowViewerForm`, который создает экземпляр объекта-формы (если этот объект не был создан ранее) и делает его видимым.

Замечание

Обращаем внимание на то, что данный метод рассчитан как на случай, когда форма создается приложением автоматически, так и на ситуацию, когда форму следует создавать "вручную".

Метод `__ShutDown` должен уничтожать объект формы только в том случае, если этот объект был создан методом `__ShowViewerForm`. Для проверки этого служит переменная `FormCreated`. Перед тем как вывести форму просмотра справки, метод `__ShowViewerForm` присваивает свойству `FileName` объекта `TextBrowser1` имя файла справки, соответствующее значению свойства `HelpFile` элемента управления, вызвавшего справку. Получить имя файла справки можно с помощью метода `GetHelpFile` интерфейса `IHelpManager`. Следует отметить, что разные элементы управления одного и того же приложения могут использовать разные справочные файлы.

Метод `UnderstandsKeyword` класса `THTMLHelpViewer` всегда возвращает значение 1. Для объекта `HelpSystem` это означает, что для каждого ключевого слова определен один раздел справки (что соответствует структуре нашего справочного файла, описанной выше).

Метод `GetHelpStrings` возвращает имя раздела для заданного ключевого слова. Напомним, что в нашей модели ключевое слово и имя раздела совпадают.

Метод `CanShowTableOfContents` всегда возвращает `true`, это означает, что в файле справочной системы всегда должно присутствовать содержание (раздел `toc`).

Методы `ShowTableOfContents` и `ShowHelp` отвечают за отображение соответствующего раздела справочного файла. Сначала метод `ShowHelp` вызывает метод `__ShowViewerForm`, гарантирующий, что форма `HelpViewerForm` существует, видима и что в объект `TextBrowser1` загружен необходимый файл справки. Поскольку каждый раздел справочной системы идентифицируется статической "привязкой" (anchor), для его вывода используется метод `ScrollToAnchor` объекта `TextBrowser1`.

Форма вывода справки для запущенного приложения показана на рис. 18.2. Создав подобный модуль для просмотра контекстной справки, мы, скорее всего, захотим использовать его во всех создаваемых нами приложениях. Для большего удобства форму просмотра справки (и соответствующий модуль) можно разместить в репозитории объектов. Для этого, в режиме разработки, щелкнем правой кнопкой мыши в окне формы `HelpViewerForm` и в контекстном меню выберем команду **Add to Repository...**. В открывшемся после этого диалоговом окне мы можем указать название нового элемента репозитория, его описание, имя автора, выбрать страницу диалогового окна **New Items**, на которой будет отображаться элемент (рекомендуется выбрать страницу **Forms**) и указать его пиктограмму. После этого, для того чтобы

добавить средство просмотра контекстной справки в новый проект, достаточно будет выбрать соответствующий пункт на странице **Forms** диалогового окна **New Items**. В проект будет автоматически добавлен новый модуль. Отметим еще раз, что, поскольку объекты этого модуля не взаимодействуют непосредственно с другими объектами приложения, новый модуль не нужно указывать в разделе `uses` других модулей. Все, что нам остается сделать, это назначить имена справочных файлов свойствам `FileName` элементов управления нашего приложения (обычно достаточно присвоить имя справочного файла свойствам `FileName` объектов-форм, используемых в приложении) и назначить каждому элементу управления ключевые слова, соответствующие разделам справочной системы, которые описывают этот элемент.

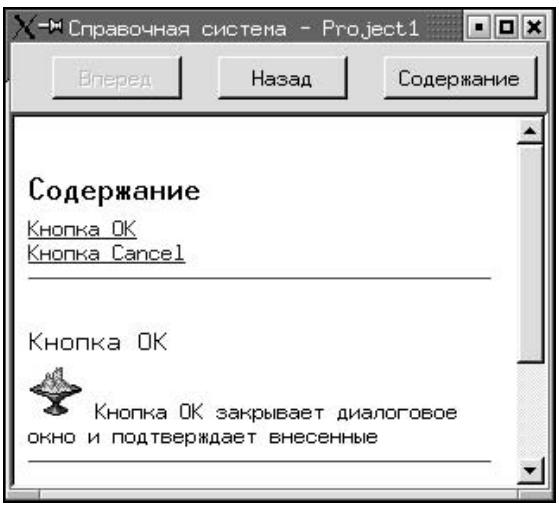


Рис. 18.2. Форма вывода контекстной справки для приложения

Kylix-приложения и разделяемые библиотеки

Как уже отмечалось в главе 1, невозможность экспорта классов из разделяемых библиотек создает дополнительные проблемы взаимодействия Kylix-приложений с библиотекой Qt library. В предыдущих версиях Kylix эта проблема решалась за счет использования двух разделяемых библиотек, поставляемых вместе со средой разработки: библиотека libqt.so.2.3.0, содержащая классы Qt library для Kylix CLXDisplay API и libqtintf-6.9-qt2.3.so, реализующая сам интерфейс CLXDisplay API. Распространять Kylix-приложения было необходимо вместе с этими двумя библиотеками, что создавало определенные неудобства, так как библиотека libqt.so.2.3.0 могла конфликтовать с библиотекой Qt library, поставляемой вместе с дистрибутивом ОС Linux.

В Kylix 3 появилась новая библиотека libborqt-6.9.0-qt2.3.so. Эта библиотека полностью инкапсулирует Qt library и интерфейс CLXDisplay API и призвана облегчить распространение Kylix-приложений отдельно от среды разработки. Библиотека libborqt-6.9.0-qt2.3.so не конфликтует с системными библиотеками ОС Linux, так что ее можно смело размещать в каталоге /usr/lib и запускать приложения Kylix без какой-либо дополнительной настройки. Впрочем, Kylix-приложения можно заставить работать и по прежней схеме. Для этого в случае с Delphi IDE следует установить переменную окружения `CLX_USE_LIBQT`, а в случае с C++ IDE использовать директиву `#define CLX_USE_LIBQT`.

Разумеется, в зависимости от специфики нашего приложения нам может понадобиться распространять вместе с ним и другие библиотеки из поставки Kylix (например, драйверы DbExpress для приложения баз данных). Консольное приложение ОС Linux может вообще не нуждаться в библиотеках из набора Kylix.

Если наше приложение использует и другие интерфейсы ОС Linux, ему понадобится связь с другими библиотеками системы. По целому ряду причин рекомендуется использовать динамическую загрузку библиотек с помощью функции `dlopen` *(см. гл. 1)*. Одной из причин этого является то, что в ОС Linux многие библиотеки могут размещаться в разных каталогах. Например, если библиотека компилируется из исходных текстов, она часто размещается в каталоге /usr/local/lib, в то время как та же библиотека, устанавливаемая из дистрибутива ОС Linux, может быть размещена в каталоге /usr/lib. Отсутствие нужной символической ссылки на библиотеку также может стать причиной того, что приложение "не найдет" библиотеку. Преимущество динамической загрузки библиотек заключается в том, что, если библиотека не найдена, приложение может либо выполнить "интеллектуальный" поиск библиотеки в других каталогах, либо "цивилизовано" проинформировать пользователя о том, какая библиотека не найдена. При статическом связывании библиотек, если библиотека не найдена, приложение выводит сообщение только на консоль и тут же завершается, что может смутить пользователя программы, особенно неопытного.

Распространение Kylix-приложений

В этом разделе мы рассмотрим вопросы создания Make-файлов для распространения приложений в форме исходных текстов и создания дистрибутивов для скомпилированных приложений.

Утилита make является частью мощного набора инструментов, предназначенного для облегчения распространения машино- и кроссплатформенных-приложений. Сборка приложений с помощью утилиты make фактически является стандартом в ОС Linux и FreeBSD. Однако необходимость в make-файлах для Kylix-приложений может вызвать определенные сомнения.

Действительно, главная функция make заключается в организации сборки проектов, состоящих из большого числа исходных файлов. Но управление проектами Kylix реализовано в рамках самого средства разработки. Так зачем нам нужна утилита make?

Есть три причины, побуждающие Kylix-программистов использовать Make-файлы. Во-первых, утилита make позволяет выполнять чрезвычайно быстро сборку приложений из уже готовых исходных текстов. Ввести одну команду проще, чем запускать графическую IDE. Во-вторых, использование Make-файлов дает пользователям нашей программы, собирающим ее из исходных текстов, привычный интерфейс, стандартный для ОС Linux. Распространение Kylix-программ в виде исходных текстов выглядит не так уж нереально, если учесть, что компилятор dcc, которым мы воспользуемся, входит в состав бесплатного Kylix Open Edition. В-третьих, многие утилиты ОС Linux рассчитаны на работу с утилитой make. Это касается, например, менеджера пакетов rpm. Если мы собираемся распространять свое приложение в rpm-пакетах, Make-файлы будут нам весьма полезны.

Как работает утилита make? После запуска make ищет в каталоге, в котором она была запущена, файл с названием Makefile и выполняет его. Make-файлы очень похожи на файлы сценариев оболочки. Так же, как и в файлах оболочки, в Make-файлах можно определять переменные и задавать последовательности команд. Основное отличие Make-файлов заключается в наличии в них нескольких разделов (target rules). Если команда make дана без параметров, утилита выполняет раздел all соответствующего Make-файла. В качестве параметра команде `make` можно указать имя раздела, который следует выполнить. Внутри раздела можно вызывать другие разделы, подобные вызовам подпрограмм. Каждый раздел состоит из объявления и описания. Объявление включает в себя имя раздела и исходные параметры (в англоязычных руководствах — targets). В качестве исходных параметров могут выступать имена разделов или файлов. Описание раздела содержит команды, выполняемые в разделе.

Make-файлы для языка Delphi Language

Утилита make — инструмент командной строки. Для компиляции с помощью make мы воспользуемся строчным компилятором dcc (это, пожалуй, единственная ситуация, в которой dcc действительно нужен). При работе с dcc, возникает та же проблема переменных окружения, что и с Kylix IDE. Перед запуском компилятора следует установить значения нескольких переменных. Обычно, эти операции выполняет скрипт kylixpath, запускаемый перед запуском компилятора, однако выполнение скрипта kylixpath в Make-файле не даст никакого эффекта. Причина этого в том, что каждая команда Make-файла выполняется утилитой make в отдельном вызове shell. Чтобы

обойти эту проблему нам придется создать дополнительный скрипт для запуска компилятора, содержащий следующие команды:

```
source $1/bin/kylixpath
$1/bin/dcc -B -U$1/lib -O$1/bin $2
```

Присвоим файлу скрипта имя build.sh и статус исполнимого файла:

```
chmod build.sh +x
```

Теперь можно написать Make-файл.

В листинге 18.3 приводится текст файла Makefile для сборки Kylix-приложения Viewer (файл должен быть размещен в каталоге проекта).

Листинг 18.3. Make-файл для приложения Delphi Language

```
#---------------------------------------------
# Viewer application makefile
#---------------------------------------------
#Присвойте переменной kylixpath путь к каталогу Kylix в вашей системе:
kylixpath=/home/user/kylix3
all: build clean
build: pas2html.dpr
       ./build.sh $(kylixpath) Viewer.dpr
clean:
       rm *.dcu
install:
       mv Viewer /usr/local/bin
```

Переменная `kylixpath` должна содержать путь к каталогу, в котором установлен Kylix в нашей системе. В различных системах Kylix может быть установлен в разных каталогах, при этом никаких глобальных переменных, содержащих путь, программа не устанавливает. По этой причине пользователям, собирающим нашу программу при помощи make, придется редактировать Make-файл вручную.

Раздел `all` состоит из одного объявления, в котором вызываются разделы `build` и `clean`. Раздел `build` (сборка) выполняет скрипт build.sh, передавая ему в качестве параметров значение переменной `kylixpath` и имя файла проекта. В качестве исходного параметра в этом разделе задано имя файла проекта. В случае с Kylix это просто формальность, но вообще за этим стоит очень полезная возможность утилиты make. Перед выполнением раздела make проверяет, изменились ли целевые файлы раздела с момента предыдущего вызова make. Если файлы не изменились, раздел не выполняется. Это позволяет сократить время сборки приложений в процессе отладки, когда изменения вносятся только в отдельные файлы проекта. Строки описа-

ния раздела, содержащие команды, должны начинаться с символа табуляции. Раздел `clean` (очистка) удаляет из каталога проекта файлы, созданные в процессе компиляции. Имена разделов `build` и `clean` не являются обязательными. Раздел `install` выполняет установку приложения. Таким образом, утилиту make можно использовать по стандартной схеме:

```
$ make
$ su
<пароль>
# make install
```

Можно также выполнить один раздел `build`:

```
make build
```

В этом случае .dcu файлы не будут удалены.

Make-файлы для языка C++

Kylix C++ IDE содержит специальную команду, позволяющую генерировать Make-файлы для консольного компилятора bc++ (команда меню **Project|Export Makefile...**). В процессе использования Make-файлов для языка C++ мы сталкиваемся с той же проблемой, что и для языка Delphi Language — проблемой переменных окружения, только в случае с языком C++ нам придется пойти другим путем. Если в случае языка Delphi Language мы создавали скрипт, вызывавшийся из Make-файла, в случае языка C++ мы напишем скрипт, вызывающий утилиту make. Допустим, в C++ IDE у нас есть проект, называемый Viewer. Создадим с помощью указанной выше команды Make-файл, который по умолчанию будет назван Viewer.mak. Чтобы выполнить этот файл с помощью утилиты make, нам понадобится специальный скрипт (назовем его startmake), приведенный в листинге 18.4.

Листинг 18.4. Скрипт для вызова утилиты make

```
#!/bin/bash
#Присвойте переменной BCB путь к каталогу Kylix в вашей системе:
BCB=/home/user/kylix3
export BCB

# BEGIN STRING TABLE

KYDEF_LOCALE="en_US"
LC_ALL_IS_C1="The LC_ALL environment variable is set to C.  Kylix cannot
start with this setting."
LC_ALL_IS_C2="Defaulting LC_ALL to"
```

```
# END STRING TABLE

if [ -z "$LANG" ]; then
   LANG=$KYDEF_LOCALE
   export LANG
fi

if [ "$LC_ALL" = "C" ]; then
   echo "$LC_ALL_IS_C1"
   echo "$LC_ALL_IS_C2 $KYDEF_LOCALE."
   LC_ALL=$KYDEF_LOCALE
   export LC_ALL
fi

export LD_LIBRARY_PATH=$BCB/bin/mozilla:$LD_LIBRARY_PATH
export MOZILLA_FIVE_HOME=$HOME/.borland/borpreview

source $BCB/bin/kylixpath $BCB >/dev/null
make -f $*
```

Мы видим, что наш скрипт является видоизмененной версией скрипта для запуска Kylix C++ IDE. Переменная `BCB` используется в самом Make-файле и содержит путь к каталогу, в котором расположена IDE. Очень удобно разместить эту переменную в нашем скрипте (из которого она будет доступна и Make-файлу). Таким образом, мы сразу указываем всем утилитам путь к каталогу Kylix IDE. Скрипт startmake можно использовать с любым Make-файлом, сгенерированным Kylix C++ IDE, передавая имя Make-файла в качестве параметра. Например:

```
./startmake Viewer.mak
```

Дистрибутивы Kylix-приложений

На сегодняшний день стандартным средством установки приложений в ОС Linux является менеджер пакетов RPM (RPM, RedHat Package Manager). Если мы работаем в ОС Linux, значит, нам наверняка известно, какие возможности менеджер RPM предоставляет *пользователям* системы Linux, и останавливаться на этом мы не будем. Опишем процесс создания дистрибутива Kylix-приложения с помощью менеджера RPM-пакетов.

Пакеты приложений можно собирать как из исходных текстов (при этом в процессе сборки пакета выполняется компиляция и сборка приложения), так и из готовых двоичных файлов. В первом случае нам потребуются исходные тек-

сты приложения. Желательно (хотя и необязательно), чтобы исходные тексты сопровождались Make-файлом и были упакованы в архив tar.gz. В этом разделе будет описан процесс создания RPM-пакета на основе уже скомпилированных двоичных файлов приложения. Как установка приложения из пакета, так и сборка пакетов приложений выполняются с помощью консольной утилиты rpm, с которой мы наверняка имели дело в ходе работы с ОС Linux.

Для того чтобы собрать пакет, нужно разместить его компоненты в определенных каталогах. Корневой каталог для сборки RPM-пакетов в ОС Linux Red Hat — /usr/src/redhat/, а в ОС Linux Mandrake — /usr/src/RPM/.

Корневой каталог содержит несколько подкаталогов, в которых и следует размещать компоненты будущего пакета:

- /SOURCES/ — в этом каталоге размещаются исходные тексты приложения, которые будут компилироваться в процессе сборки пакета (поскольку мы будем собирать дистрибутив из уже скомпилированных файлов, этот каталог нам не понадобится);
- /SPECS/ — в этом каталоге размещаются spec-файлы, управляющие процессом сборки пакетов;
- /BUILD/ — в этом каталоге хранятся временные файлы, необходимые в процессе сборки;
- /RPMS/ — в этот каталог программа rpm помещает готовые пакеты двоичных файлов (данный каталог содержит ряд подкаталогов, соответствующих различным моделям процессоров, и, если не указано иное, готовый пакет будет помещен в подкаталог, соответствующий процессору, который установлен на нашем компьютере);
- /SRPMS/ — в этот каталог программа rpm помещает готовые пакеты исходных текстов.

Инструкции по сборке пакетов содержатся в специальных файлах (spec-файлы). Далее мы рассмотрим структуру spec-файла для сборки дистрибутива некоего приложения Viewer.

Spec-файл состоит из нескольких разделов, первым из которых является *преамбула* (preamble). Преамбула содержит общие сведения о пакете, такие как имя пакета, номер версии и релиза, данные о поставщике и упаковщике программного обеспечения. Кроме этого, в преамбуле можно задать некоторые важные опции, влияющие на весь процесс сборки пакета. Преамбула состоит из нескольких тэгов (tags), которым присваиваются соответствующие значения. Формат записи тэгов в преамбуле выглядит так:

```
имя_тэга: значение
```

Преамбула может содержать множество различных тэгов, большая часть которых не является обязательными. Наиболее часто используемые тэги перечислены в табл. 18.2.

Таблица 18.2. *Тэги-преамбулы spec-файла*

Тэг	Значение
`Summary`	Краткое описание содержимого пакета
`Name`	Имя пакета
`Version`	Версия приложения
`Release`	Номер сборки пакета
`Copyright`	Сведения о лицензии на программное обеспечение
`Group`	Произвольное название группы базы данных RPM, в которую следует поместить информацию о пакете
`Source`	Ссылка на исходные тексты приложения (если они имеются)
`URL`	Адрес сайта распространителя пакета
`Vendor`	Сведения о производителе программного обеспечения
`Packager`	Сведения о создателе пакета
`Requires`	Список пакетов, которые должны быть установлены до установки данного пакета
`Distribution`	Название дистрибутива ОС Linux, для которого собирается пакет
`BuildRoot`	Корневой каталог для каталогов двоичных файлов приложения
`Prefix`	Подкаталог корневого каталога, в котором размещаются двоичные файлы приложения
`DocDir`	Каталог, содержащий стандартную документацию приложения (относительно корневого каталога)

Значения тэгов `Name`, `Version` и `Release` используются программой rpm при генерации имени файла пакета. Кроме тэгов, преамбула spec-файла, как и любой другой его раздел, может содержать переменные. Это — одна из многих черт, роднящая spec-файлы с файлами сценариев оболочки.

За преамбулой следуют остальные разделы spec-файла, первым из которых обычно является раздел `%description`, содержащий "длинное" описание пакета, которое может состоять из нескольких строк. Далее следует раздел `%prep`, содержащий команды оболочки ОС Linux, выполняющие различные подготовительные операции, необходимые для сборки пакета (создание временных каталогов, распаковка архивных файлов и т. п.).

Если процесс сборки пакета совмещается со сборкой приложения, в spec-файле создается раздел `%build`, содержащий инструкции, необходимые для сборки (это может быть просто вызов утилиты make).

Далее идет раздел `%clean`, который выполняется в самом конце процесса сборки, после записи готовых rpm-файлов на диск. В этом разделе должны содержаться команды, удаляющие все временные файлы и каталоги, созданные в процессе сборки пакета.

Следующие разделы `%post` и `%postun` представляют собой post-install и post-uninstall скрипты. В отличие от рассмотренных выше разделов spec-файла, эти команды выполняются не в момент сборки пакета, а во время его установки на машине пользователя. Команды, содержащиеся в этих разделах, будут выполняться, соответственно, после установки и после удаления пакета из системы. Например, если пакет содержит разделяемые библиотеки, указанные разделы могут содержать вызов утилиты `ldconfig`, конфигурирующей систему после установки/удаления библиотек. В общем случае сюда можно включать любые операции, например, создание ярлыков приложения, настройку файлов конфигурации под конкретную систему и т. п.

Совет

Для того, чтобы деинсталляция приложения выполнялась чисто, необходимо придерживаться следующего правила "все, что делается в разделе `%post`, должно быть отменено в разделе `%postun`".

Кроме разделов `%post` и `%postun` в spec-файлы можно включать разделы `%pre` и `%preun`, выполняемые, соответственно, перед установкой и удалением пакета. Например, в раздел `%pre` можно включить подготовительные действия, необходимые при установке приложения поверх уже установленной версии. К разделам, выполняющимся на машине пользователя, относится также раздел `%verify`, позволяющий проверить правильность установки пакета.

Заключает spec-файл раздел `%files`, являющийся, пожалуй, самой важной его частью. Этот раздел содержит список двоичных файлов и файлов документации, которые следует поместить в пакет. Список файлов может включать групповые символы (wildcards). Следует также помнить, что информация о каталогах, в которых расположены файлы, также будет включена в пакет, и при инсталляции приложения пользователем соответствующие файлы будут размещены в тех же каталогах, в которых они находились при сборке пакета. Изменить этот порядок можно с помощью тэга `BuildRoot`, о чем будет сказано ниже.

Возможно, описание структуры spec-файла показалось нам слишком сложным. Однако на деле все не так страшно, как кажется. Рассмотрим пример spec-файла для сборки пакета приложения Viewer viewer.spec (листинг 18.5).

Листинг 18.5. Файл viewer.spec

```
#
# Spec-файл для сборки пакета приложения Viewer
#
```

```
Summary: Прекрасное приложение Viewer.
Name: Viewer
Version: 1.2.3
Release: 1
Copyright: GPL/LGPL
Group: Graphic Applications/Misc
URL: http://www.some.site.some.domain/viewer/
Vendor: I. Sidorov, inc.
Packager: Ivan Sidorov, ivan@some.site.some.domain
%description
Это приложение предназначено ...

%post -p /sbin/ldconfig

%postun -p /sbin/ldconfig

%files
/usr/local/bin/viewer
/usr/lib/libviewer.so.1.2.3
```

Если разместить файл viewer.spec в каталоге SPECS корневого каталога системы RPM и затем дать команду:

```
rpm -ba viewer.spec
```

в одном из подкаталогов каталога RPMS появится пакет viewer-1.2.3-1.arch.rpm, где "arch" — сокращенное обозначение типа процессора, соответствующее имени подкаталога. Пакет будет содержать файлы /usr/local/bin/viewer и /usr/lib/libviewer.so.1.2.3.

Фактически spec-файл для сборки пакета двоичных файлов приложения может состоять только из преамбулы и раздела `%files`.

А для чего же нужен тэг `BuildRoot`? В рассмотренном выше примере, двоичные файлы, предназначенные для установки в системе пользователя в каталогах /usr/local/bin/ и /usr/lib/, копируются в пакет из одноименных каталогов нашей системы. Однако, допустим, мы расположили файлы приложения во временных каталогах, например, /var/tmp/viewer/usr/local/bin/ и /var/tmp/viewer/usr/lib/. При этом мы хотим, чтобы в системе пользователя эти файлы устанавливались в подкаталогах корневого каталога /usr. Для того, чтобы решить эту проблему, нужно присвоить тэгу `BuildRoot` значение /var/tmp/viewer/. После этого при сборке пакета в разделе `%files` ссылки на файлы будут считаться не абсолютными, а относительными (подкаталогами каталога `BuildRoot`). При установке пакета ссылки из раздела `%files` будут считаться абсолютными (подкаталогами корневого каталога системы).

Глава 19

Приложения для электронного бизнеса

В этой главе мы опишем принципы построения приложений, предназначенных для ведения бизнеса в Интернете. Несмотря на высказанный в последнее время скептицизм относительно перспектив интернет-бизнеса, описанные ниже технологии продолжают развиваться, пусть и не такими быстрыми темпами, как представлялось в начале.

Основные понятия

Приложения электронного бизнеса можно разделить на две основные категории:

1. Intra-business-приложения.
2. Inter-business-приложения.

Первая категория включает приложения, ограниченные рамками компании/организации и связанные с внутрикорпоративной деятельностью. К этой категории относятся, например, интранет-серверы, предоставляющие информацию сотрудникам предприятия. Для приложений этого типа характерна модель **Бизнес-Сотрудник** (B2E, business to employee). Приложения этой архитектуры предоставляют сотрудникам предприятия доступ к внутренним базам данных и другим информационным ресурсам и позволяют им публиковать собственные информационные ресурсы и обмениваться сообщениями друг с другом.

Вторая категория включает приложения, предназначенные для организации взаимодействия между предприятием и внешними агентами, такими как потребитель и другие предприятия. Эти приложения, в свою очередь, могут принадлежать одной из нескольких моделей электронного бизнеса.

Модель **Бизнес-Бизнес** (B2B, business to business) — вид деятельности, в ходе которой две компании проводят бизнес-транзакции с помощью Интернета. Например, компания может разместить запрос на коммерческие предложения, получить текущие котировки от своих поставщиков, заклю-

чить контракт, получить или оплатить счета, опубликовать документы. Данный вид деятельности позволяет построить на новом уровне коммерческие связи между производителями и создает благоприятную почву для партнерских отношений между ними. Рынок становится более открытым, и появляется масса возможностей взаимодействия и приобретения необходимых товаров и услуг. Система B2B может быть как открытой, т. е. доступной для обычных пользователей Глобальной сети или других возможных бизнес-партнеров, так и закрытой — предназначенной только для определенных партнеров или рабочих групп и исполняющей определенные технологические функции.

По функциональным возможностям сайты B2B можно разделить на следующие группы:

1. *Каталоги.* Эти сайты являются наиболее простым и распространенным вариантом системы B2B, с помощью которой покупатели находят продавца товарных позиций с фиксированной ценой.
2. *Электронные биржи.* Отличаются большей сложностью по сравнению с каталогами и функционально подобны реальным биржам. Используются в основном для торговли товарами широкого потребления.
3. *Аукционы.* Такие сайты функционально подобны реальным аукционам и виртуальным аукционам B2C (*см. ниже*), но в силу специфики модели B2B часто используются для продажи излишков запасов.
4. *Электронные сообщества.* Создаются для решения нетрадиционных бизнес-задач, таких как разного рода исследования, политическое лоббирование или обмен идеями. Интернет является идеальной средой для объединения усилий и интересов.

В качестве дальнейшего развития этой модели определенное распространение получила модель **Бизнес-Электронный рынок-Бизнес** (B2M2B business to market to business), где в качестве посредника между двумя системами выступает электронная торговая площадка — интернет-портал, облегчающий поиск партнеров и заключение сделок.

Модель **Бизнес-Потребитель** (B2C business to consumer) на сегодняшний день является наиболее популярной формой электронной коммерции. В рамках этой модели деятельность компании нацелена на прямые продажи конечному потребителю. При решенных проблемах с доступом в Интернет в регионах, с надежной работой платежных систем и служб доставки, модель B2C может стать эффективным средством устранения различий между крупными городами и удаленными регионами в смысле доступности товаров и услуг для потребителя. Еще один плюс B2C — прямые продажи с минимальным количеством посредников. Устранение посредников дает возможность устанавливать конкурентные цены на местах и даже увеличивать их (исключая процент посредников), что естественно приведет к росту прибыли.

Модель **Потребитель-Потребитель** (C2C, consumer to consumer) позволяет организовать продажу товаров и услуг между потребителями интернет-компании. В данном случае сайт выступает в роли посредника между покупателем и продавцом.

Модель **Потребитель-Бизнес** (C2B, consumer to business) предоставляет потребителю возможность устанавливать стоимость различных товаров и услуг, предлагаемых компаниями, по собственному выбору. Этот вид электронной коммерции является наименее развитым по сравнению с остальными. В качестве примера можно привести американскую компанию (**www.priceline.com**), которая дает возможность своему покупателю назвать цену, за которую он хотел бы купить товар или услугу. Таким образом, формируется спрос, который не означает, что совершится продажа по запрошенной цене. Продавец, пользуясь данными текущего спроса, принимает окончательное решение. Сайт модели C2B выступает в роли посредника-брокера в попытке найти продавца, готового продать товар или услугу по цене, сформированной предложениями покупателей.

Рынок бизнес-средств для Интернета постоянно развивается, но, несмотря на непрерывное возникновение новых направлений и методов их реализации, все они соответствуют одной из вышеописанных моделей.

Структура решений архитектуры B2C

Прежде чем приступить к рассмотрению основных типов структур (топологий) систем электронного бизнеса модели B2C, следует упомянуть еще одно понятие, связанное с этой моделью. Речь идет о разделении корпоративных приложений на Front End-приложения и Back End-приложения. Back End-приложения осуществляют непосредственный доступ к данным предприятия и выполняют другие операции в рамках заданной системы электронного бизнеса. Front End-приложения осуществляют такие операции, как взаимодействие с клиентским программным обеспечением, авторизация удаленных пользователей, управление их индивидуальными ресурсами и обработка клиентских запросов. Схема корпоративной системы электронного бизнеса с выделенными участками Back End и Front End показана на рис. 19.1.

Архитектура B2C допускает различные типы структуры (топологии). Мы выделим две простейшие структуры — двухуровневую и трехуровневую. Выбор между этими архитектурами определяется необходимостью использования в программных комплексах приложений сторонних производителей. Если приложения сторонних производителей не требуются, выбирается двухуровневая архитектура, в противном случае — трехуровневая.

Двухуровневая архитектура представлена на рис. 19.2.

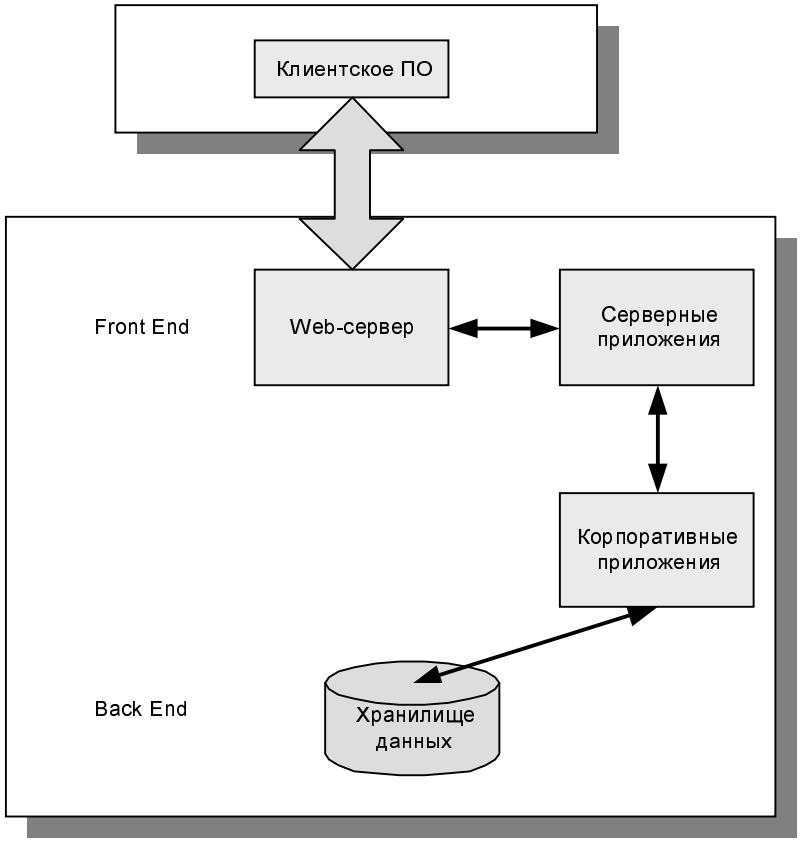


Рис. 19.1. Front End и Back End-приложения

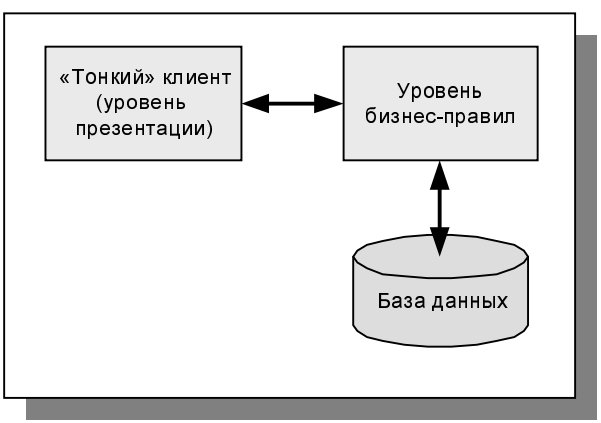


Рис. 19.2. Двухуровневая архитектура приложения B2C

В рамках двухуровневой системы бизнес-правила реализованы на уровне сервера Web-приложений. Информация, необходимая для авторизации пользователя, может быть получена с использованием службы каталогов или службы безопасности. Информация, необходимая пользователям, извлекается из базы данных, расположенной в интра-сети предприятия. В этой архитектуре возможны варианты. Например, в целях повышения безопасности сервер Web-приложений может быть перенесен за брандмауэр домена.

Трехуровневая структура комплекса B2C применяется при наличии в системе приложений сторонних производителей. Схема подобной структуры представлена на рис. 19.3.

В рамках трехуровневой системы бизнес-правила могут быть реализованы на уровне сервера Web-приложений и на уровне приложений третьей стороны. Информация, необходимая для авторизации пользователя, может быть получена с использованием службы каталогов или службы безопасности. Информация, необходимая пользователям, извлекается из базы данных, расположенной в интра-сети предприятия.

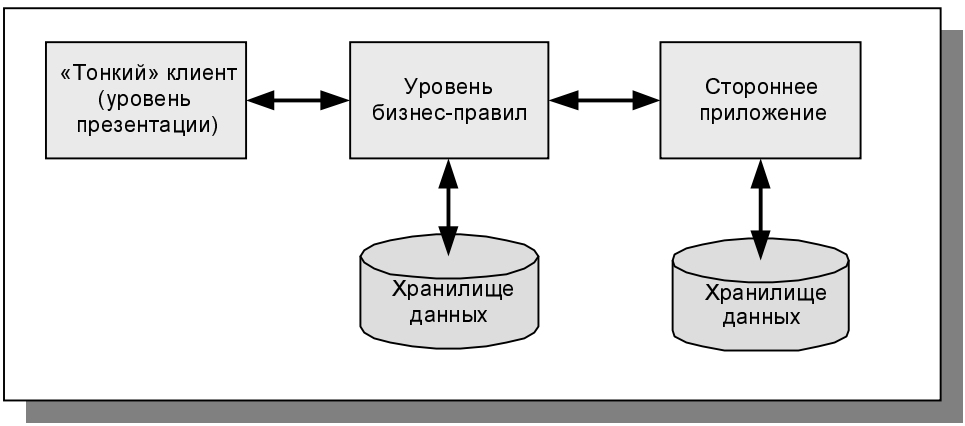


Рис. 19.3. Трехуровневая архитектура приложения B2C

Структура решений архитектуры B2B

В этом разделе мы рассмотрим три категории систем архитектуры B2B:

- системы обмена документами;
- системы непосредственного обмена данными с помощью адаптеров и мостов;
- брокеры сообщений.

Системы обмена документами

Бизнес-партнеры могут прибегнуть к этой топологии для повышения эффективности обмена данными. Архитектура системы этого типа, осуществляющей обмен документами между двумя компаниями, показана на рис. 19.4.

Приложения обмениваются сообщениями по заранее согласованному протоколу. Важной характерной чертой систем этого типа является наличие буферов сообщений, позволяющих накапливать сообщения для последующей передачи их блоками (batches), что особенно удобно, если для обмена данными используется арендованный канал. При приеме блоков сообщений они также накапливаются в буфере принимающей стороны, откуда приложение, обрабатывающее сообщения на принимающей стороне, извлекает их по мере обработки.

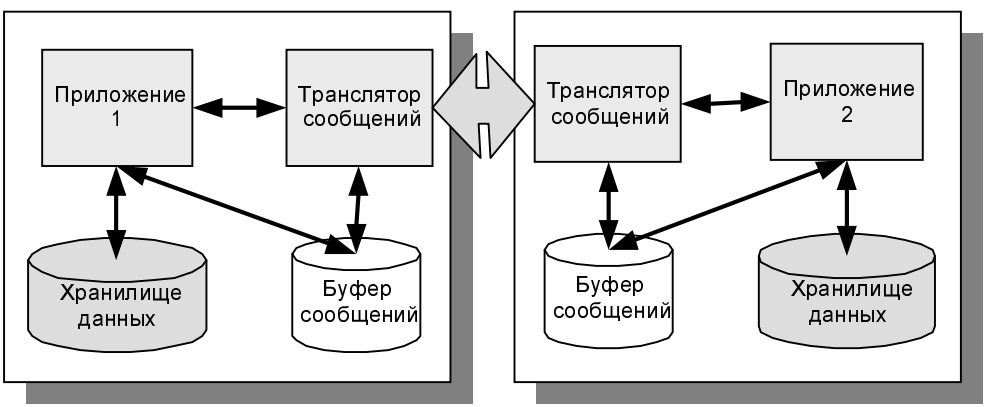


Рис. 19.4. Топология системы обмена документами

Системы непосредственного обмена данными

Эти системы используются в тех случаях, когда обмен данными осуществляется на достаточно низком уровне, позволяющем преобразовать (с помощью адаптеров или мостов-преобразователей) сообщения, содержащие данные, в вызовы функций интерфейса приложений. Примером такой системы могут служить приложения, обменивающиеся данными по протоколу SOAP. Топология системы этого типа представлена на рис. 19.5.

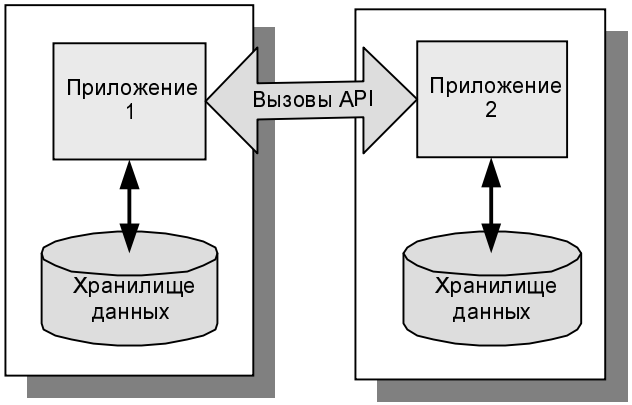


Рис. 19.5. Приложения, использующие прикладной интерфейс для обмена данными

Брокеры сообщений

В рамках данной топологии одно или несколько корпоративных приложений представляется удаленному клиенту в виде набора служб (services).

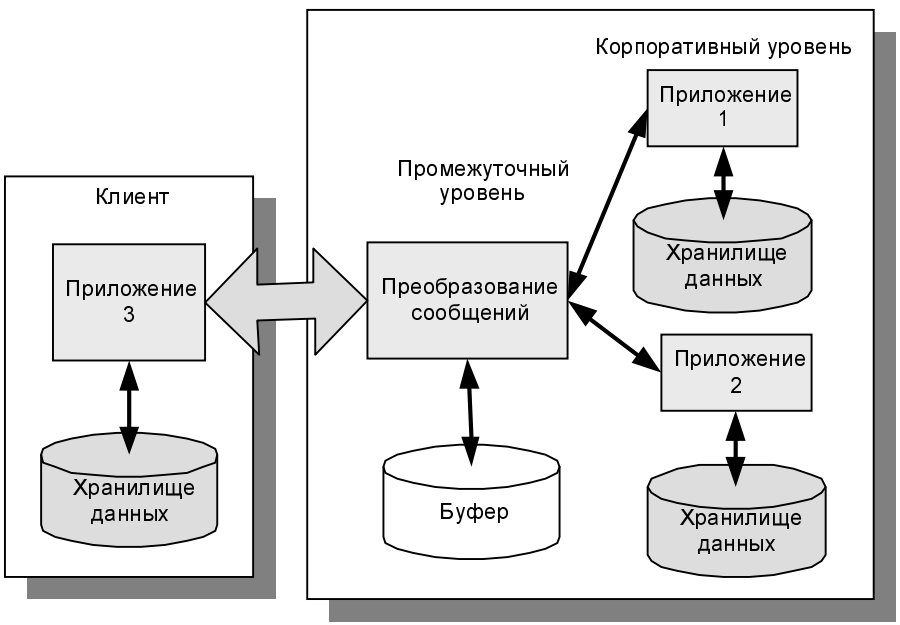


Рис. 19.6. Система, использующая брокер сообщений

Роль промежуточного уровня в рамках этой топологии выполняет брокер сообщений, осуществляющий анализ, маршрутизацию и преобразование сообщений, которыми клиент обменивается с корпоративным сервером. Схема этой топологии показана на рис. 19.6.

Возможность предоставления корпоративной стороной множества сервисов и связанная с этим необходимость обработки большого числа сообщений требуют наличия на промежуточном уровне, где выполняются преобразования между системой сообщений и системой сервисов, специального буфера.

Заключение

На протяжении всей этой книги неоднократно говорилось о том, что даже в рамках перечня рассмотренных тем изложение не претендует на всеохватность. Многие технологии, реализованные в Borland Kylix, могли бы стать темами отдельных книг. Сегодняшние средства разработки от Borland — это не просто средства создания программ для разных операционных систем, а наборы решений, позволяющих программистам использовать самые передовые технологии в таких областях, как Интернет и приложения баз данных. Естественно, что и книга, описывающая программирование в такой среде, тем более книга, рассчитанная на читателя, уже знакомого с основами программирования, должна затрагивать не только приемы программирования, но и основные принципы соответствующих технологий. Одной из замечательных особенностей сред разработки от Borland является то, что эти средства облегчают не только процесс программирования приложений с использованием новых технологий, но и процесс изучения этих технологий. Речь здесь идет не столько о встроенной справочной системе, сколько о самой интегрированной среде визуальной разработки, которая делает применение новых компонентов настолько наглядным, что позволяет программисту изучать их возможности методом эксперимента. В связи с этим нельзя не отдать должное той изобретательности, с которой разработчики Kylix включают новые технологии в традиционную схему визуальной разработки. Благодаря этим усилиям, программист может изучать практическое применение новых технологий, не покидая привычной для него среды разработки.

Автор, со своей стороны, надеется, что ему удалось выполнить поставленную перед собой задачу — познакомить читателя с передовыми технологиями разработки, реализованными в Borland Kylix, многие из которых являются новинкой в мире операционной системы Linux.

Еще до того, как компания Borland официально объявила о намерении выпустить "Delphi для ОС Linux", слухи об этом начали циркулировать в программистском сообществе. Реакция некоторых программистов, связанных с ОС Linux, оказалась тогда неожиданной. На многих форумах и конференциях, посвященных Linux-программированию, можно было услышать утверждения о том, что полноценная реализация среды Delphi для ОС Linux в мире ОС Linux не только не нужна, но и невозможна. Аргументы приводились самые разные, начиная с обилия наборов графических компонентов в

ОС Linux, из-за чего Borland, утверждали критики, не сможет выбрать "правильную" основу для своих визуальных компонентов, и заканчивая отсутствием в ОС Linux таких, широко используемых в Delphi технологий, как ActiveX и COM/DCOM.

Примечание

Надо сказать, что относительно основы для набора графических компонентов у компании в то время действительно не было ясности. Первоначальные пресс-релизы Borland сообщали, что среда визуальной разработки для ОС Linux будет основана на gtk — открытом и бесплатном наборе графических элементов, на которых построена оболочка GNOME. Однако окончательный выбор был сделан в пользу Qt library.

Однако сегодня, после выхода уже третьей версии Kylix — полноценного средства разработки с ярко выраженной идеологией и четко очерченными тенденциями дальнейшего развития, не только вопрос о жизнеспособности такого средства в мире ОС Linux, но и вопрос о его необходимости, решается сам собой.

Многие поклонники ОС Linux, в том числе и автор этой книги, особенно ценят одно из отличительных качеств этой операционной системы — ее открытость. Для нас, безусловно, важно, что ОС Linux не просто бесплатная операционная система с общедоступными исходными текстами. ОС Linux является "живым" доказательством того, что интернет-сообщество независимых программистов способно реализовать крупный проект, основанный на принципах, совершенно отличных от принципов гигантских компаний, ориентированных исключительно на коммерческие продукты. Однако в связи с этим не следует впадать и в противоположную крайность, свойственную некоторым поклонникам ОС Linux — пренебрежительно-недоверчивое отношение к коммерческим продуктам, создаваемым для этой ОС.

Примечание

При том, что Kylix был и остается коммерческим продуктом, компанию Borland нельзя упрекнуть в неуважении к традициям Linux-программирования. Доказательством соблюдения этих традиций является разработка Kylix Open Edition, который, хотя и лишен многих функций коммерческих версий, уверенно догоняет их, например, в плане качества редактора текста, что особенно заметно в последней версии.

Наверное, что такое отношение к коммерческим Linux-проектам лишено разумных оснований. Сама система Linux не перестанет быть открытой и бесплатной оттого, что кто-то выпускает для нее закрытые и коммерческие приложения. Все не коммерческие проекты, разрабатываемые для ОС Linux под лицензией GPL, так и останутся некоммерческими. При этом хочется задать всем сторонникам ОС Linux, неодобрительно отзывающихся о разработке коммерческих программ, один вопрос: желаете ли вы, чтобы наша

любимая операционная система сохранила свою жизнеспособность? Если ответ на этот вопрос положительный, то должно признать, что разработка коммерческих приложений для ОС Linux не ослабляет, а, наоборот, укрепляет позиции этой системы в целом. Для крупных компаний, без поддержки которых успешное развитие ОС Linux вряд ли возможно, важным фактором является наличие в рамках этой системы готовых решений, соответствующих идеологии бизнеса, принятой в этих компаниях. Не секрет, что зачастую именно коммерческие продукты, в том числе продукты Borland, предоставляют такие решения.

Итак, имея версию Borland Kylix 3, мы можем четко представить себе будущие направления развития этого средства разработки, тем более, что это развитие сохраняет преемственность не только относительно продуктов Borland, предназначенных для ОС Linux, но и относительно средств разработки из мира ОС Windows (многие тенденции, нашедшие воплощение в последней версии Kylix, получили свое начало еще в Delphi 3).

Мы с полным правом можем ожидать дальнейшего развития технологий разработки интернет-приложений, особенно технологий, связанных со стандартом XML, так как сам этот стандарт и его производные находятся в состоянии активного развития. Не останутся без внимания и технологии баз данных. Подходы, применяемые компанией Borland к этим технологиям, менялись уже не раз, и каждый раз компания представляла более элегантные и мощные решения.

Нельзя не отметить, что технологии разработки распределенных приложений, предоставляемые средой Borland Kylix, соответствуют принятым на сегодняшний день методам корпоративного программирования, и, несомненно, будут эволюционировать вместе с этими методами. Объединение средств разработки на языках Delphi Language и C++ в один пакет также является весьма перспективным решением, поскольку увеличивает степень свободы разработчика, позволяя использовать преимущества каждого из языков программирования в рамках одного комплексного решения и даже в рамках одного проекта.

Любой программист, следящий за новинками в области средств разработки для ОС Linux, не может не отметить, что такие среды разработки, как Borland Kylix и Delphi, инспирировали появление и стимулировали развитие множества проектов, реализующих аналогичные подходы к программированию. Сам этот факт может служить доказательством существования и перспективности направлений, выбранных разработчиками компании Borland.