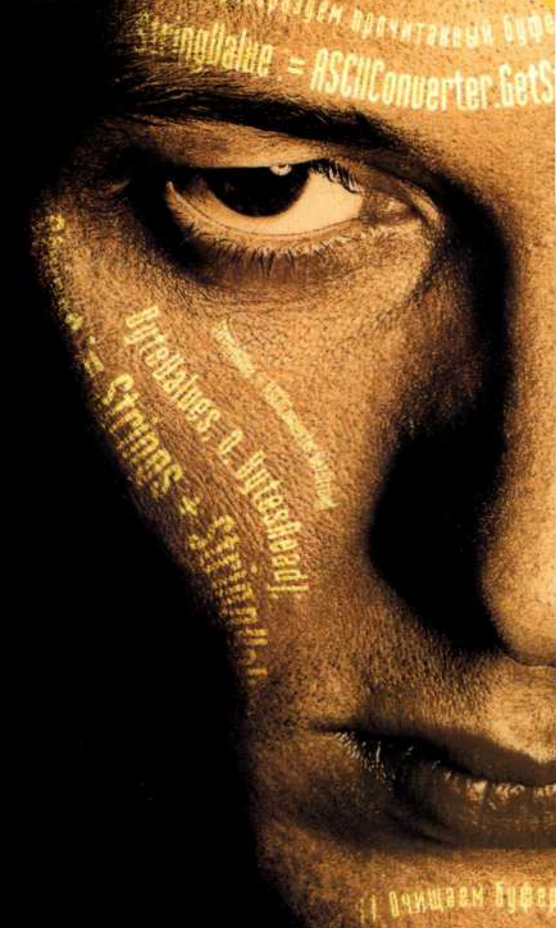
Михаил Флёнов

Delphi 2005

СЕКРЕТЫ ПРОГРАММИРОВАНИЯ

CD-ROM прилагается



- Алгоритмы на Delphi
- Эффективное использование баз данных
- Готовые решения типичных задач

Михаил Флёнов

Delphi 2005

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2006

ББК 32.973-018
УДК 004.42
Ф71

Флёнов М. Е.

Ф71 Delphi 2005. Секреты программирования (+CD). — СПб.: Питер, 2006. — 266 с.: ил.

ISBN 5-469-01164-X

В книге описываются практические приемы программирования на Delphi, решения реальных задач, с которыми сталкиваются программисты в ежедневной работе. Автор не ограничивается рассмотрением какой-то определенной технологии — он стремится обучить читателя профессиональному программистскому мышлению. Издание будет полезно тем, кто хочет более глубоко изучить язык программирования Delphi и его возможности. Для понимания представленного материала достаточно начальных знаний по языку программирования Delphi.

ББК 32.973-018
УДК 004.42

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5-469-01164-X

© ЗАО Издательский дом «Питер», 2006

Краткое содержание

Введение 10

Глава 1. Знакомство с Delphi 2005 14

Глава 2. Советы и секреты 38

Глава 3. Базы данных 101

Глава 4. Алгоритмы 183

Глава 5. Управление проектами 250

Заключение 264

Содержание

Введение 10

Замечания по работе с книгой 11

Благодарности 12

От издательства 13

Глава 1. Знакомство с Delphi 2005 14

Встречаем по одежке 15

Редактор кода 19

Проверка правописания 20

Рефакторинг 21

Переименование 21

Объявление переменной 22

Объявление поля 23

Создание метода 23

Поиск модуля 25

Подсказки 26

История изменений 27

Помощь 28

Новые компоненты 30

Компонент PopupActionBar 30

Компонент CategoryButtons 30

Компонент ButtonGroup 33

Базы данных 33

Синхронное редактирование 35

Поиск метода 36

Поиск компонента 37

Управление файлами 37

Глава 2. Советы и секреты . . . 38
Секреты DFM . . . 38
Расширение возможностей стандартных компонентов . . . 41
Доступ к защищенным свойствам и методам . . . 44
Сохранение данных проекта . . . 45
Общие сведения о формате XML . . . 53
XML в Delphi . . . 61
Анализ документа с помощью компонента TXMLDocument . . . 61
Атрибуты тега . . . 67
Изменение XML-документа . . . 67
Редактирование . . . 70
Сохранение изменений . . . 71
Домашнее задание . . . 71
XML и бинарные данные . . . 71
Список открывавшихся файлов . . . 72
Работа с «горячими» клавишами . . . 77
Красивые меню . . . 82
Коллекции . . . 88
Работа с сеткой StringGrid . . . 91
Секреты и проблемы StringGrid . . . 91
Сохранение и загрузка данных . . . 96

Глава 3. Базы данных . . . 101
Постановка задачи . . . 102
Реализация справочников . . . 106
Фильтрация . . . 108
Редактирование данных . . . 110
Работа с клиентами . . . 114
Суперпоисковые поля . . . 116
Управление запросами . . . 123
Сортировка . . . 125
Эффективный поиск . . . 129
Шаблон клиентского окна . . . 135
Трехуровневые приложения . . . 136
Основы трехуровневых приложений . . . 136
Создание сервера . . . 138

Создание клиента 141
Выборка и изменение данных 143
Секреты работы с сервером бизнес-логики 147
Реализация шаблонов 147
Просмотр измененных данных 148
Обработка ошибок сохранения 150
Сторонние разработки 151
Хранение запросов 151
Передача параметров 152
Создание универсальных окон 153
Косметика 153
Выделение строк с помощью закладок 154
Выделение строк через ключ 157
Позиционирование 160
Экспорт в Word 161
Создание документа Word 161
Вывод текста 162
Позиционирование 163
Шрифт 165
Размещение текста 166
Работа с таблицами 168
Расширение возможностей 170
Данные кубиком 172
Глава 4. Алгоритмы 183
RTTI 184
Постановка задачи 184
Реализация дизайнера 185
Определение свойств компонента 190
Сохранение свойств компонента 194
Графика 197
Оптимизация графики на примере построения градиента 197
Попиксельный анализ изображения 203
Цветовая панель 206
Манипуляции с компонентом 215
Разработка универсального менеджера компонентов 215
Использование универсального менеджера компонентов 225
Резюме 226
Отображение файлов в памяти 226

Печать 231
Сохранение содержимого компонента TTreeView в файле 234
Сохранение содержимого компонента TTreeView в базе данных 243

Глава 5. Управление проектами 250

Хранение проекта 250
Моделирование 252
Моделирование для Win32 252
Моделирование для .NET 257
Резюме 262

Заключение 264

Введение

Недавно я зашел в книжный магазин и обратил внимание на достаточно большое число книг, призванных учить программированию начинающих программистов. Каждый автор делает это по-своему, но в большинстве книг даются только основы. Описать все, что можно сделать с помощью той или иной среды разработки, в одной книге практически невозможно, поэтому любая книга по Delphi закладывает только фундамент.

А что делать тем, кто уже знает основы и научился создавать приложения средней сложности? Для таких программистов выбор книг уже не столь велик. В основном это книги, посвященные определенной технологии, например программированию звука, графики, баз данных и т. д.

В данной книге мы не будем ограничиваться одной технологией. Наша задача — повысить свой уровень как программистов и узнать что-то новое. В 2004 году в издательстве БХВ уже вышла моя книга «Библия Delphi», тоже предназначенная для начинающих и тоже описывающая только основы, хотя я и пытался сделать ее более профессиональной. Так что теперь я не буду останавливаться на основах. Мы двинемся дальше и начнем более глубокое изучение Delphi на примере последнего детища Borland — Delphi 2005.

Но вначале познакомимся с новой средой разработки Delphi 2005. Рассмотрим новые возможности визуального редактора и редактора кода, а также новые компоненты. Одним из важных нововведений этой версии я бы назвал удобные функции для рефакторинга кода, и на этом вопросе мы остановимся достаточно подробно.

После этого мы перейдем к знакомству с секретами программирования, позволяющими обходить ограничения компонентов Delphi и делать еще многое другое. Вас ожидает большое количество примеров программ на Delphi, интересные алгоритмы и решения типовых задач, с которыми вы можете столкнуться в реальной жизни (по крайней мере мне подобные задачи решать приходится достаточно часто). Я надеюсь, что эти примеры помогут вам в вашей работе.

Большое количество примеров направлено на то, чтобы показать вам, как сделать код более эффективным. Нет, я не буду затрагивать тему оформления кода и визуального интерфейса. Это я уже сделал в книге «Delphi в шутку и всерьез:

что умеют хакеры», вышедшей в издательстве «Питер» в 2004 году. Основной упор мы будем делать на практике и алгоритмах, поэтому вас ждет интересный код.

Итак, для понимания темы книги вам необходимо обладать начальными знаниями о программировании в среде Delphi и иметь опыт работы с Delphi более ранних версий. Так как в основном мы будем рассматривать логику и алгоритмы, основные функции вы должны уже знать. Материала, представленного в книге «Библия Delphi», для этого вполне достаточно.

Ваши вопросы, пожелания и комментарии я жду по адресу horrific@vr-online.ru, а также на форуме сайта www.vr-online.ru. Чтобы быстрее получить ответ, я рекомендую использовать форум, который я посещаю каждый день и где регулярно общаюсь с читателями. Я жду любых откликов по теме книги; они очень важны для меня, поскольку все мои работы основаны на общении с читателями, на их вопросах и пожеланиях.

Замечания по работе с книгой

Книга состоит из пяти глав:

1. **Знакомство с Delphi 2005.** Здесь вы познакомитесь с новыми возможностями среды разработки Delphi версии 2005.
2. **Советы и секреты.** Эта глава полностью посвящена раскрытию различных секретов. Это позволит упростить разработку приложений и уберечь вас от стрессов. Вы узнаете, как расширять возможности стандартных компонентов без создания новых потомков, как хранить данные программы в файлах, на практике познакомитесь с популярным ныне форматом хранения данных XML и многое другое.
3. **Базы данных.** В этой главе мы узнаем, как создавать эффективные приложения для работы с базами данных. Представленный здесь универсальный код сделает поддержку больших проектов максимально простой и удобной. Я приготовил множество интересных процедур и функций, которые пригодятся вам в реальной работе.
4. **Алгоритмы.** В этой главе вас ждут интересные алгоритмы, позволяющие упростить разработку программ. Основными темами, которые мы затронем в этой главе, будут RTTI (Run Time Type Information — информация о типах во время выполнения) и графика.
5. **Управление проектами.** Эта глава, как явствует из ее названия, посвящена советам по управлению проектами. Здесь вы познакомитесь с UML-моделированием, которое значительно упрощает поддержку программ, предназначенных для платформы Win32, а для платформы .NET упрощает еще и программирование. Вы узнаете, как можно создавать объекты, не написав ни одной строки кода.

Для подготовки примеров к книге я использовал Delphi 2005. Если вы работаете с более старой версией, могут возникнуть проблемы с открытием примеров с прилагаемого к книге компакт-диска. Но это не значит, что примеры вовсе не

будут работать в вашей версии. Большая часть из них будет работать, если вы самостоятельно повторите описываемые в книге действия в своей версии.

Но даже если у вас установлена среда Delphi 2005 и примеры с компакт-диска открываются, я рекомендую все описываемые действия выполнять самостоятельно. В этом случае материал будет усваиваться и запоминаться намного лучше. Исходные примеры даны для того, чтобы вы могли посмотреть на фрагменты кода, не приводимые в книге для экономии места, а также сравнить результаты работы примеров с компакт-диска и разрабатываемых самостоятельно.

Примеры на компакт-диске чаще всего содержат больше кода и более функциональны, поскольку в книге я старался приводить только самые основные фрагменты.

Раньше меня очень часто критиковали за то, что в своих примерах я не оставляю читателям возможностей для самостоятельного расширения предлагаемых программ. В данной работе я исправляюсь и часто описываю задачи, которые рекомендуется решать самостоятельно. Это позволит вам закрепить изученный материал и потренироваться в навыках программирования. Чтобы задачи проще было решать, я даю некоторые пояснения и даже фрагменты кода. Тем не менее будет неплохо, если вы попытаетесь найти свое решение. Возможно, оно окажется лучше моего.

На компакт-диске в каталоге Doc вы найдете множество дополнительных документов по программированию в формате PDF. Как всегда, я постарался сделать диск не менее интересным, чем сама книга. На диске вы найдете документы не только по программированию, но и на «околокомпьютерные» темы.

Для работы с этими документами вам понадобиться программа Acrobat Reader. Ее можно бесплатно загрузить с сайта www.adobe.com или найти на дисках, которые продают с компьютерными журналами, играми или программами. Желательно, чтобы версия была не ниже 5-й, иначе могут быть проблемы с разметкой, и документ будет выглядеть не так, как я задумывал.

Благодарности

В каждой книге я благодарю людей, которые мне помогали в ее создании. Эта книга не исключение, потому что люди, которых я благодарю, действительно помогают мне и заслуживают больше, чем просто благодарность.

Первым делом хочется поблагодарить вас, моих постоянных читателей, за то, что вы поддерживаете меня своими комментариями к книгам. Положительные отзывы дают энергию к созданию новых работ, а отрицательные заставляют думать и находить решения, позволяющие делать свою работу лучше. Все письма, которые попадают в мой почтовый ящик, я обязательно читаю и по возможности стараюсь отвечать как можно быстрее. Наиболее интересные вопросы по программированию превращаются в примеры и попадают в книги, потому что это может быть интересно и полезно многим.

Хочется поблагодарить сотрудников редакций, с которыми мне приходится работать. Редакторы помогают сделать книгу лучше, подправляют мои недочеты

и направляют мою деятельность в нужное русло. С руководителем проекта мы вместе занимаемся названием, обложкой, организационными вопросами, а это позволяет заложить фундамент успеха книги еще на этапе ее зарождения. Иногда правильно выбранное направление уже делает книгу бестселлером, и моя задача лишь следовать этому направлению. Дизайнеры и верстальщики делают книгу удобной для чтения, выделяющейся на фоне других.

Отдельное спасибо друзьям из редакции журнала «Хакер» и команде VR за их помощь и поддержку. Это два крупных источника для подпитки моих идей и продвижения моих книг.

Большое спасибо моей семье. На создание книг уходит очень много времени, ведь в книгах по программированию приходится писать не только текст книги, но и большие объемы кода, а это отнимает очень много времени. Спасибо родным за то, что терпят мои исчезновения в виртуальной реальности по вечерам и даже в выходные.

Я начал работать над книгой в начале осени 2004 года, а осенний период очень труден для котов, поэтому посильное участие в написании этой книги принял мой новый кот Чекист. Каждый вечер он начинал орать так, что мой сон благополучно улетучивался, и я мог спокойно работать допоздна. Правда, приходилось иногда отрываться от компьютера, когда рев (именно рев) становился совсем уже невыносимым, и приходилось бегать по квартире с веником, дабы успокоить Чекиста.

Прошу прощения, если я кого-то упустил, потому что хочется поблагодарить всех моих друзей, всех знакомых и вообще всех, кто мне хоть как-то помогает.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

Глава 1

Знакомство с Delphi 2005

До появления Delphi версии 4 я работал в этой среде достаточно редко и использовал только по мере надобности. В те времена чаще приходилось работать с Visual C++ или Borland C++. С появлением Delphi 4 я полюбил эту среду разработки, и она окончательно перебралась на первое место в списке моих наиболее часто запускаемых программ, потому что оказалась действительно удобной и имеет все необходимые для успешного программирования возможности. В течение года все мои проекты постепенно переместились в Delphi и продолжают в ней сопровождаться.

Переходя на все последующие версии Delphi, я только отдавал дань моде. Если бы я этого не делал, то в будущем, для того чтобы «перескочить» через версию, пришлось бы приложить намного больше усилий. Возможности, которые предоставляли новые версии, не слишком отличали их от предыдущих, и без них вполне можно было бы обойтись. В основном нововведения касались добавления нескольких компонентов для доступа к новым технологиям и небольших изменений в самой среде разработки.

Самым значимым изменением в Delphi за все эти годы можно считать только появление библиотеки CLX, которая позволяет писать код как для Windows, так и для Linux с помощью среды разработки Kylix. Но это событие имело особое значение только для тех программистов, которые разрабатывают приложения для Windows и Linux.

Появление Delphi 2005 я считаю самым грандиозным событием для каждого программиста, использующего эту среду разработки. Однако многие программисты с трудом восприняли этот переход из-за слишком большого числа изменений, к которым необходимо привыкнуть. В этой главе я постараюсь дать необходимую информацию, которая поможет тем, кто хочет перейти на среду Delphi 2005, и тем, кто уже перешел, но хочет узнать о ней больше.

Рассмотрим основные нововведения. Вы оцените всю мощь новой среды разработки. Нововведений так много, что даже для описания основ мне понадобилась целая глава.

Встречаем по одежке

Вечный вопрос — какой язык программирования лучше? Я всегда говорю, что лучше тот, который удобнее именно вам и позволяет решить поставленные задачи (мощь языка). Первое является наиболее весомым фактором и включает в себя удобный и красивый интерфейс, мощный редактор кода, удобство визуального редактора и т. д.

Мощь языка невозможно измерить. По устоявшемуся мнению, наиболее мощным является C++ и именно поэтому я начинал изучение программирования с этого языка. Но когда я увидел, что даже на Visual Basic можно создавать великолепные программы, то понял, что мощь в большей степени зависит от программиста, чем от среды разработки.

В Delphi 2005 нам действительно предлагают интерфейс, который может отпугнуть программиста. Но если поработать с ним хотя бы пару часов, вы сможете оценить мощь и удобство этой среды разработки и полюбите ее так же, как я.

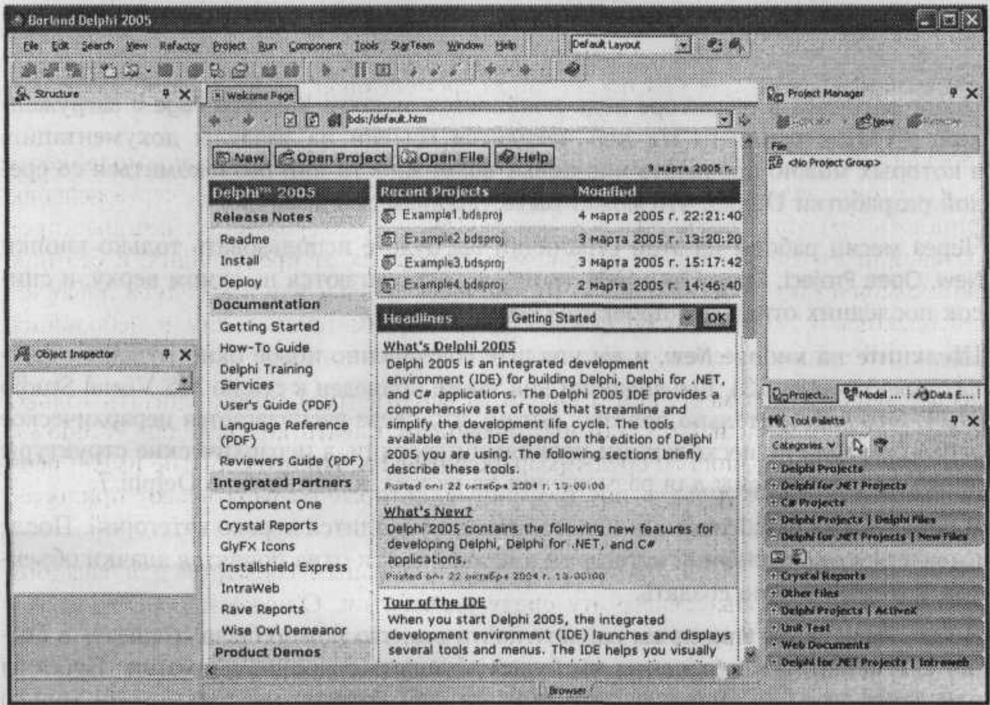


Рис. 1.1. Главное окно Delphi 2005

Главное неудобство, которое ощущается сразу же после установки, — загрузка Delphi 2005 происходит намного дольше. Это связано с тем, что в одной среде были объединены возможности разработки:

- классических программ на Delphi под Win32 (продолжение Delphi 7);
- программ на Delphi под платформу .NET (продолжение Delphi 8 .NET);
- программ на языке C# под платформу .NET.

Изменения в Delphi 2005 по сравнению с Delphi 7 и предыдущими версиями видны уже сразу же после запуска (рис. 1.1). Интерфейс стал ближе к среде разработки Visual Studio .NET от Microsoft, где все средства находятся в одном окне. На мой взгляд, это более удобное решение, и в предыдущих версиях мне приходилось самостоятельно объединять окна в одно целое, иначе можно было «заблудиться» во множестве открытых окон. В Delphi 2005 такая раскладка окон установлена по умолчанию.

Если вы считаете, что многооконный интерфейс является более удобным, вы всегда можете вернуться к нему. Для этого на панели Desktop есть раскрывающийся список, в котором можно выбрать раскладку (рис. 1.2). По умолчанию используется вариант Default Layout. Для переключения на раскладку в стиле Delphi 7 выберите пункт Classic Undocked.

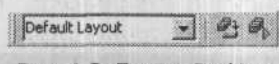


Рис. 1.2. Панель Desktop

После загрузки в редакторе кода появляется вкладка Welcome Page с загруженной HTML-страницей. На ней находятся ссылки на разделы документации, в которых можно прочитать про новые возможности или познакомиться со средой разработки Delphi, что может быть удобным для новичков.

Через месяц работы из этой странички вы будете использовать только кнопки New, Open Project, Open File и Help, которые располагаются на самом верху, и список последних открытых проектов (Recent Projects).

Щелкните на кнопке New, и вы увидите совершенно новое окно создания файла/проекта (рис. 1.3). Его внешний вид также приведен к стилю MS Visual Studio .NET. Это действительно полезно, потому что слева располагается иерархическое дерево разделов с несколькими уровнями иерархии, а иерархические структуры плохо приспособлены для размещения на вкладках, как было в Delphi 7.

В левой части окна создания нового проекта находится дерево категорий. После выделения определенной категории в правой части окна появятся значки объектов, которые можно создать.

В категории Delphi Projects вы найдете почти все, что можно было создавать в Delphi 7. Именно на эти проекты мы будем обращать основное внимание. Проекты для платформы .NET (категория Delphi for .NET Projects) мы рассмотрим только в общих чертах.

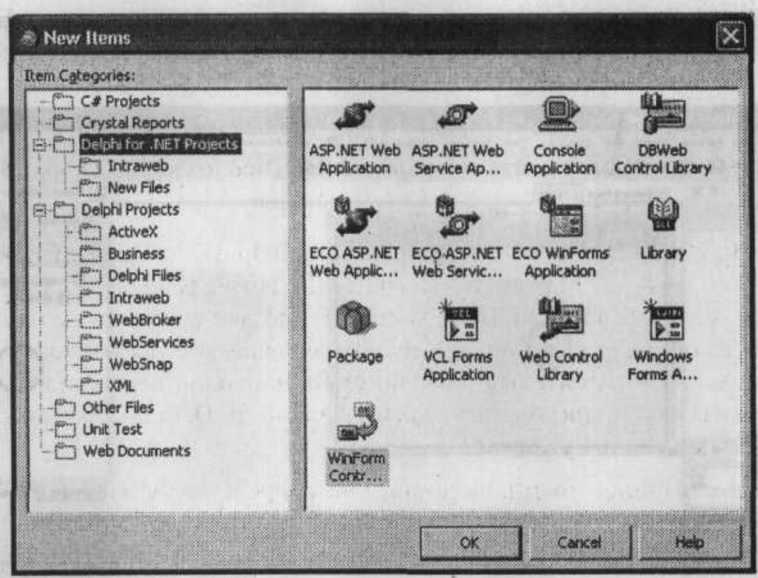


Рис. 1.3. Окно создания файла/проекта

Давайте создадим новый проект и посмотрим на обновленный дизайнер форм. Выберите в дереве категорий категорию Delphi Projects и дважды щелкните на значке VCL Forms Application в правой части окна. Перед нами появится окно дизайнера форм (рис. 1.4). Хотя его возможности и набор компонентов будут отличаться от доступных при создании других типов приложений (например, приложения .NET), основа одна и та же. То есть вы будете создавать различные типы приложений в схожей по структуре и внешнему виду среде.

Рассмотрим основные части окна.

Слева вверху (1) находится панель, в которой отображается структура компонентов на форме. В Delphi 7 эту панель заменяло окно Object TreeView.

Слева внизу (2) находится окно объектного инспектора (Object Inspector). Его внешний вид также изменился, все свойства теперь разбиты по категориям. С одной стороны, это очень удобно, но я привык искать имена свойств по алфавиту, поэтому иногда возникают проблемы. Чтобы отсортировать имена свойств по алфавиту, щелкните в панели объектного инспектора правой кнопкой мыши и в появившемся контекстном меню выберите команду Arrange ▸ By Name.

В центре окна (3) располагается форма для визуального конструирования окна будущего приложения. Здесь особых изменений нет.

Справа вверху (4) находится окно менеджера проектов (Project Manager). Здесь также нет визуальных изменений, разве что стало больше команд в контекстном меню, что делает некоторые команды более доступными.

Справа внизу (5) можно увидеть палитру компонентов (Tool Palette). Эта палитре претерпела наибольшие изменения и требует отдельного разговора. Она также обрела стиль среды Visual Studio .NET, что было отрицательно воспринято

многими программистами, но деваться некуда, приходится привыкать. Благо есть элементы управления, которые позволяют сделать эту палитру чуть более удобной.

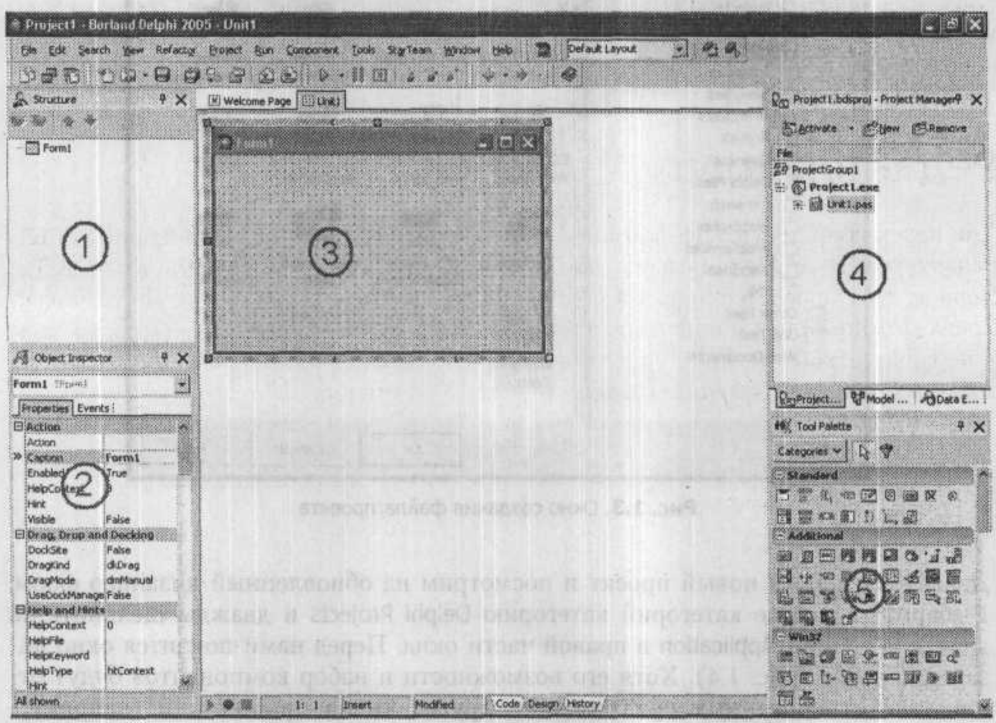


Рис. 1.4. Дизайнер форм при создании приложений WinAPI

По умолчанию на палитре компонентов компоненты отображаются в виде кнопок, объединенных по категориям с отдельными заголовками. Таким образом, каждый компонент занимает целую панель, поэтому приходится слишком много листать, чтобы найти нужную кнопку. Лучшим вариантом является запрет на отображения заголовков категорий. В этом случае компоненты будут располагаться более компактно.

Чтобы запретить отображение заголовков, проще всего щелкнуть правой кнопкой мыши на палитре компонентов и в появившемся контекстном меню выбрать команду Properties. Перед вами откроется окно свойств, в котором нужно сбросить флажок Show Captions. Можно также установить флажок Auto Collapse Categories, чтобы на палитре компонентов открытой была только одна категория кнопок. При выборе любой категории все остальные будут автоматически закрываться. В некоторых случаях это удобно, но иногда лучше отключать эту возможность. Чтобы не приходилось постоянно открывать окно свойств, в контекстном меню палитры компонентов есть одноименная команда Auto Collapse Categories.

В Delphi 2005 достаточно много категорий и, когда они все открыты, приходится долго прокручивать палитру, чтобы найти нужную категорию. Для быстрого

перехода к нужной категории можно воспользоваться кнопкой Categories в заголовке палитры компонентов.

Одно из новшеств палитры компонентов состоит в том, что компоненты теперь можно перетаскивать (Drag&Drop) на форму. В предыдущих версиях это было невозможно.

Редактор кода

Для переключения между формой и редактором кода используйте клавишу F12. Редактор кода в Delphi также претерпел значительные изменения (рис. 1.5). Первое, что бросается в глаза, — возможность сворачивать фрагменты кода. Слева можно видеть такие же квадратики со знаками + (плюс) и – (минус), как в иерархической древовидной структуре. Щелкая на них, можно сворачивать и разворачивать код процедур или объявлений методов.

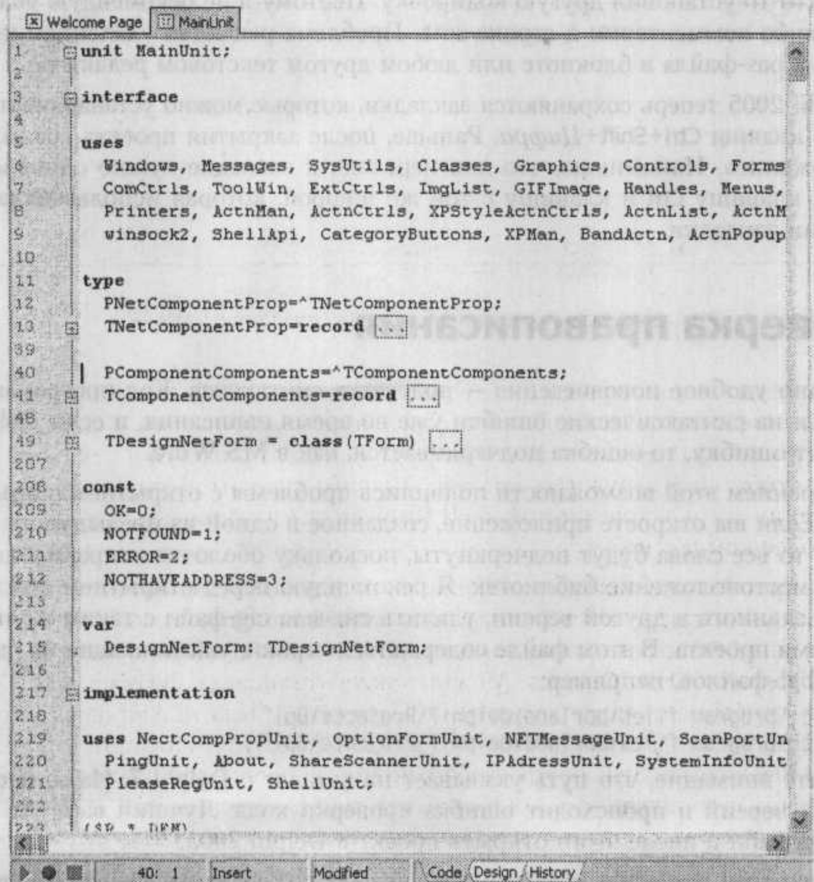


Рис. 1.5. Редактор кода

Если вы уже отработали какую-то процедуру, оптимизировали и отладили ее, то можно закрыть фрагмент ее кода, чтобы он не занимал лишнее место на экране. Вместо тела процедуры в редакторе будет отображаться только ее имя и параметры.

В Delphi 2005 появилось много удобных «горячих» клавиш. Например, теперь стало довольно просто комментировать блоки кода. Для того чтобы закомментировать блок кода, выделите его и нажмите клавиши Ctrl+/ или выберите команду Toggle Comments в контекстном меню редактора. Повторное выполнение этой операции удаляет символы комментария.

Очень важно, что код теперь может храниться в разных кодировках. Щелкните правой кнопкой мыши в редакторе кода и в меню File раскройте подменю Format. В этом подменю представлены команды задания поддерживаемых кодировок, но вы в основном будете работать только с кодировкой ANSI. Зачем тогда я об этом говорю? Видимо из-за внедрения кодировок при наличии комментария в строке `unit` (это первая строка любого модуля, в которой находится его имя) содержимое файла при открытии в редакторе кода становится нечитабельным, как будто кто-то установил другую кодировку. Поэтому я не рекомендую указывать какие-либо комментарии в строке `unit`. Проблема решается удалением комментария из pas-файла в блокноте или любом другом текстовом редакторе.

В Delphi 2005 теперь сохраняются закладки, которые можно устанавливать с помощью клавиш Ctrl+Shift+*Цифра*. Раньше, после закрытия проекта все закладки уничтожались. Напоминаю, что для перехода к закладке нужно одновременно нажать клавишу Ctrl и клавишу с той же цифрой, которая использовалась при создании закладки.

Проверка правописания

Еще одно удобное нововведение — подсветка синтаксиса. Код программы проверяется на синтаксические ошибки уже во время написания, и если Delphi обнаружит ошибку, то ошибка подчеркивается, как в MS Word.

С внедрением этой возможности появились проблемы с открытием старых проектов. Если вы откроете приложение, созданное в одной из предыдущих версий Delphi, то все слова будут подчеркнуты, поскольку оболочка неправильно определяет местоположение библиотек. Я рекомендую перед открытием файла проекта, созданного в другой версии, удалить сначала cfg-файл с таким же именем, как и имя проекта. В этом файле содержаться строки, указывающие на расположение bpl-файлов, например:

```
-LE"c:\program files\borland\delphi7\Projects\Bpl"
-LN"c:\program files\borland\delphi7\Projects\Bpl"
```

Обратите внимание, что путь указывает на каталог с Delphi 7. Из-за несовместимости версий и происходит ошибка проверки кода. Лучший вариант — удалить cfg-файл и после этого открыть проект в Delphi 2005.

При открытии проектов, созданных в предыдущих версиях, Delphi спрашивает, для какой платформы преобразовать исходный код — для Delphi .NET или

Delphi Win32? Выберите нужный вариант и щелкните на кнопке OK. В проект будут автоматически внесены необходимые изменения (в основном, в конфигурационные файлы), и будет создан новый cfg-файл.

Рефакторинг

Еще одно мощное нововведение в Delphi 2005 — меню Refactor. В этом меню можно найти несколько команд, которые действительно упрощают управление кодом. Рассмотрим примеры.

Переименование

Допустим, вам необходимо переименовать переменную. Если эта локальная для процедуры переменная, то с переименованием не возникнет проблем. А если это имя формы, которая используется во множестве других форм? В таких случаях я всегда изменял имя в объявлении, а потом компилировал код, чтобы компилятор в сообщениях об ошибках показал мне места, где осталось старое имя.

Теперь проблема решается двумя щелчками мыши. Сначала в любом месте выделяем имя переменной, которую нужно переименовать, а затем выбираем команду Refactor ▸ Raname. Перед вами появится окно диалога, в котором необходимо указать новое имя (рис. 1.6).

Рис. 1.6. Окно задания нового имени

В этом окне можно увидеть три поля:

- Unit — имя текущего модуля;
- Old name — старое имя переменной;
- New name — новое имя переменной.

Укажите новое имя и щелкните на кнопке OK. Внизу окна редактора кода появится панель Refactorings (рис. 1.7). На ней в виде дерева можно увидеть модули и строчки кода, в которых найдено имя переименовываемой переменной. Жирным шрифтом отображается текст, который предполагается изменить. Просмотрите и убедитесь, что среда Delphi верно определяет строки, которые должны быть изменены.

Чтобы согласиться с предлагаемыми Delphi изменениями в коде, необходимо нажать клавиши Ctrl+R или щелкнуть на кнопке Refactor.

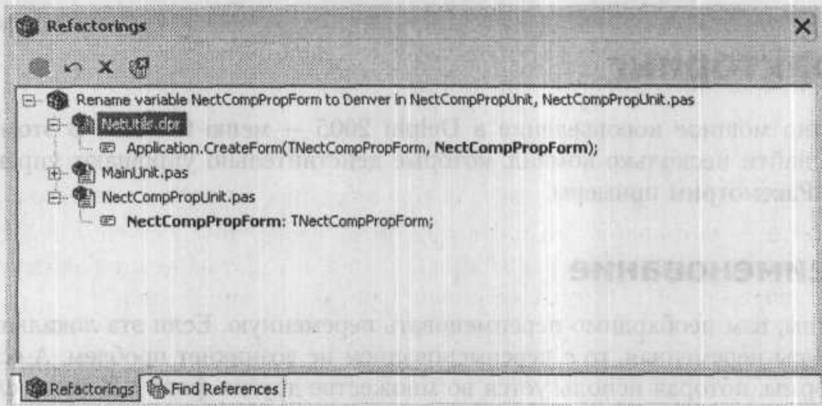


Рис. 1.7. Панель Refactorings

Объявление переменной

Допустим, вы пишете большую процедуру, которая не умещается на экране, и в самом конце нужно написать следующий цикл:

```
for i:=1 to 10 do
begin
  // Код цикла
end;
```

Допустим, переменная i не объявлена. Раньше приходилось просматривать код текущей процедуры в поисках раздела var. В таких случаях недобрым словом поминались разработчики языка Pascal, на основе которого создавался язык Delphi. В новой версии уже нет такой проблемы, потому что Borland заботится о программистах. Теперь можно просто написать этот цикл, не задумываясь об объявлении переменной i, а затем выделить ее и выбрать команду Refactor ▸ Declare Variable. Перед вами откроется окно объявления новой переменной (рис. 1.8).

Рис. 1.8. Окно объявления переменной

Рассмотрим элементы управления, доступные в этом окне.

- Name — если переменная была выделена верно, изменять имя переменной в этом поле не придется.
- Type — среда Delphi 2005 достаточно интеллектуальна и в большинстве случаев определяет тип переменной верно, но иногда бывает необходимо подкорректировать тип переменной в этом поле.
- Array — установите этот флажок, если необходимо объявить массив указанного типа.
- Dimensions — если создается массив, то в этом поле можно указать его размер.
- Set Value — установите этот флажок, если необходимо, чтобы переменной было задано значение по умолчанию. Delphi автоматически добавит код после ключевого слова `begin`, изменяющий значение переменной.

Объявление поля

Путем объявления переменной можно создать только локальную переменную. Если вы хотите создать переменную в объявлении объекта (то есть создать поле), следует использовать команду Refactor ▸ Declare Field. Перед вами откроется окно объявления поля (рис. 1.9).

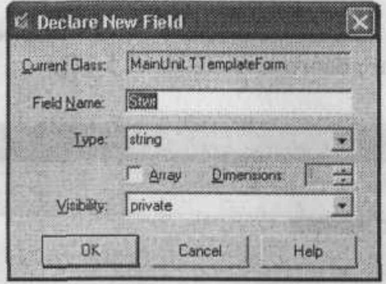


Рис. 1.9. Окно объявления поля

Большинство элементов управления в этом окне вам уже знакомы, их назначение такое же, как в окне создания переменной, но есть и два новых:

- Current Class — в этом поле отображается текущий класс;
- Visibility — в этом списке выбирается область видимости.

Создание метода

Допустим, вы пишете метод и обнаруживаете, что какой-то фрагмент кода вашего метода повторяется в другом методе. Вполне логично вынести этот фрагмент в отдельную процедуру. Раньше приходилось все делать вручную, а теперь Delphi 2005 все делает «в два щелчка» и очень быстро.

Рассмотрим листинг 1.1. В данном примере в цикле от 1 до 10 изменяется заголовок окна (поле `Caption` формы). В качестве заголовка устанавливается текст

«Заголовок» плюс значение счетчика i. Подобный цикл иногда используется для отображения хода выполнения операции в заголовке окна. Это очень удобно для пользователя — наблюдать за ходом работы приложения по заголовку окна, свернутого в кнопку на панели задач.

Листинг 1.1. Пример метода

```
var
  i: Integer;
  sTemp: string;
begin
 //Код метода
 sTemp:='Заголовок-';
 for i:=1 to 10 do
 begin
  Caption:=sTemp+IntToStr(i);
 end;
 //Код метода
end;
```

Теперь представим, что мы хотим выделить цикл в отдельный метод. Для этого выделяем цикл:

```
for i:=1 to 10 do
 begin
  Caption:=sTemp+IntToStr(i);
 end;
```

Затем выбираем в меню команду Refactor ▸ Extract Method. Перед вами откроется окно задания параметров нового метода (рис. 1.10).

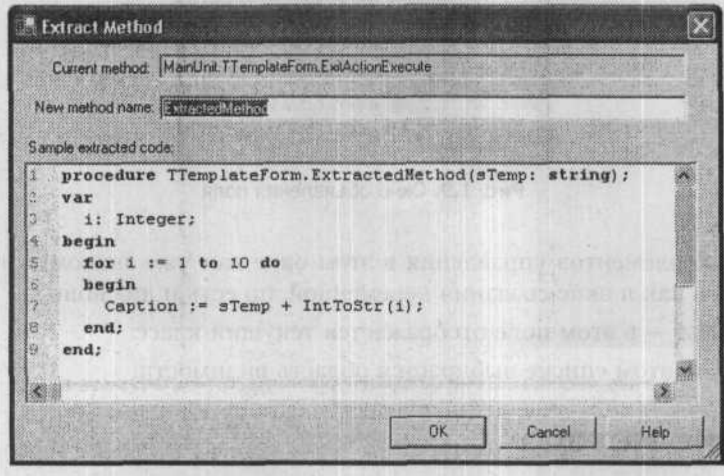


Рис. 1.10. Задание параметров нового метода

В поле Current method показано имя текущего метода. В поле New method name вы должны задать имя создаваемого метода. В редакторе Sample extracted code выводится код, который будет выделен в качестве отдельного метода. Обратите внимание, что среда Delphi автоматически определила, что переменная i должна

быть создана локально, а переменная sText — передаваться в виде параметра метода, поскольку переменная i задается внутри выделенного фрагмента, а переменная sText нет. Если бы мы выделили не только цикл, но и строку, в которой переменной sText присваивается значение, то обе переменные были бы созданы локальными.

После щелчка на кнопке OK Delphi автоматически создает новый метод, а выделенный ранее фрагмент заменяется вызовом этого метода.

Поиск модуля

Читатели очень часто спрашивают, почему код, взятый из моих книг, не работает. Чаще всего ошибка связана с отсутствием объявлений необходимых модулей. Допустим вы написали следующий код:

```
if not InputQuery('Внимание', 'Введите число', sTemp) then
  exit;
```

При компиляции Delphi выведет сообщение об ошибке E2003 Undeclared identifier: 'InputQuery', которое говорит о том, что Delphi не знает идентификатора InputQuery. Проблема в том, что функция InputQuery описана в модуле, который не подключен к текущему модулю, и для решения проблемы достаточно его подключить. Как найти модуль? В Delphi 2005 появилась удобная возможность для поиска необходимого модуля и подключения его из любой точки. Для этого выберите команду Refactor ▸ FindUnit. Перед вами откроется окно поиска модуля (рис. 1.11).

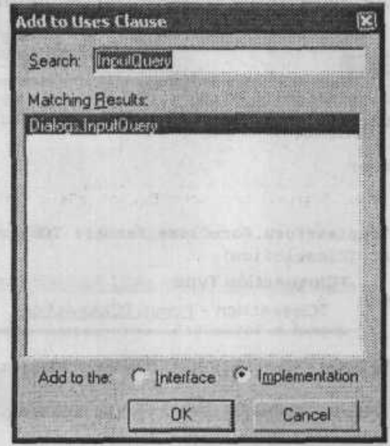


Рис. 1.11. Окно поиска модуля

В поле Search необходимо указать имя метода, типа или константы, которое необходимо найти. В списке Matching Results будут отображаться найденные модули, в которых есть искомый идентификатор. Внизу окна есть группа переключателей, с помощью которых вы можете выбрать, в какой раздел добавить объявление модуля:

- Interface — объявление будет добавлено в раздел uses после ключевого слова `interface` (это объявление располагается в самом начале модуля до описания типов);
- Implementation — объявление будет добавлено в раздел uses после ключевого слова `implementation` (этот раздел располагается после описания типов модуля).

Команду Refactor ▸ FindUnit можно выполнять, находясь в любом месте модуля. Таким образом, нет необходимости переходить в самое начало модуля в поисках раздела uses.

Подсказки

Подсказки в редакторе кода стали более красивыми и интеллектуальными. Перейдите в редактор кода и нажмите клавиши Ctrl+Пробел. Перед вами появится раскрывающийся список с именами всех доступных для ввода в данном контексте полей и методов. Если какое-то имя объекта уже набрано и поставлена точка, то в списке будут перечислены свойства и методы только этого объекта. Если теперь набрать имя нужного свойства, то количество значений в списке сократится, отображая только подходящие значения. Когда нужная строка оказывается выделенной, достаточно нажать клавишу Enter, и Delphi вставит выделенное имя поля/метода в текущую точку ввода.

Эта возможность существует в Delphi уже давно, однако в Delphi 2005 при выделении пункта в раскрывающемся списке появляется подробная подсказка. Как показано на рис. 1.12, такая же подсказка появляется, если навести указатель мыши на имя метода, переменной, константы, записи и т. д.

```
137   FIniFile.WriteInteger('Option', 'FileToolBarTop', FileToolBar.
138   FIniFile.WriteBool('Option', 'FileToolBarVisible', FileToolBar
139
140   FIniFile.Free;
141  end;
142
143  procedure TTemplateForm.FormClose(Sender: TObject;
144    var Action: TCloseAction);
145  begin            TCloseAction Type
146   SaveProgParam     TCloseAction - Forms.TCloseAction
147  end;
148
149  procedure TTemplateForm.ShowHint(Sender: TObject);
150  begin
151   StatusBar1.Panels.Items[1].Text := Application.Hint;
152
153  end;
154
155  end.
```

Рис. 1.12. Новая подсказка в редакторе кода

Благодаря более интеллектуальным подсказкам нет необходимости каждый раз открывать справочную систему.

История изменений

В предыдущих версиях Delphi существовала только одна возможность увидеть исходный код таким, каким он был до последнего сохранения, — открыть файл с расширением .bak. Однако каждое следующее сохранение стирает предыдущее, поэтому если дважды подряд сохранить исходный код, содержимое bak-файла уже не будет нести никакой полезной информации. В связи с этим я сам отключал режим создания резервных копий и всем советовал делать то же самое, чтобы не расходовать лишнее место на диске и не «замусоривать» каталог с исходным кодом.

Если вы перед переводом своего проекта на Delphi 2005 не отключали этот режим, можете удалить все bak-файлы, потому что больше они не нужны. В новой версии этой среды разработки есть более мощное средство отслеживания истории. Внизу панели редактора кода есть три вкладки:

- Code — исходный код программы;
- Design — редактор форм;
- History — новая вкладка, обеспечивающая управление историей изменений (рис. 1.13).

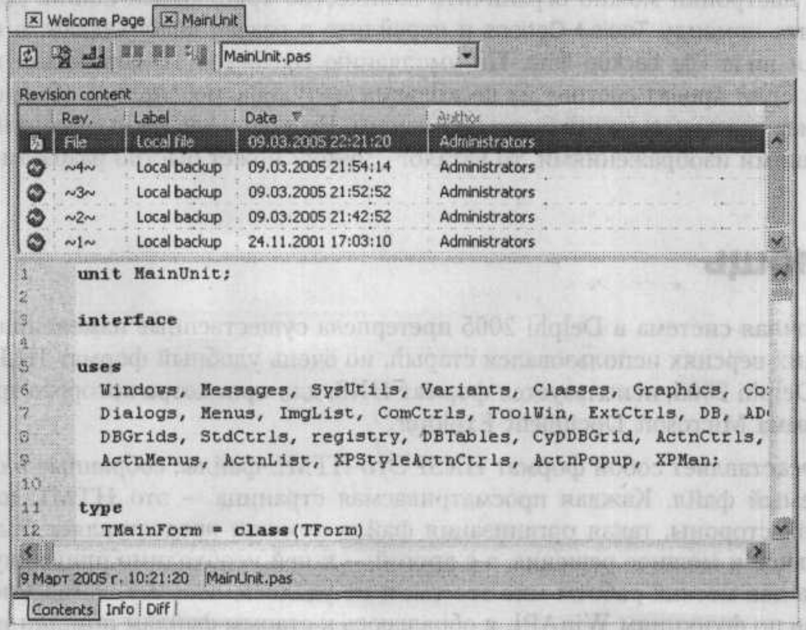


Рис. 1.13. Окно управления историей изменений

Вверху окна отображается список всех созданных резервных копий. Выбрав копию, ориентируясь на нужную дату, в центре окна в редакторе можно увидеть содержимое файла резервной копии.

Внизу окна можно увидеть три вкладки:

- Contents — содержимое резервной копии;
- Info — информация о резервной копии;
- Diff — строки, которые были добавлены, изменены или удалены с момента создания выделенной резервной копии. В этом режиме на панели инструментов окна истории изменений становятся доступными кнопки перехода на следующую или предыдущую запись (Next difference и Previous difference), в которой найдено отличие. Одноименные команды имеются и в контекстном меню.

Если вы хотите восстановить какую-то резервную копию, то выделите ее и щелкните на кнопке Revert to previous revision (вернуться к предыдущей копии) или выберете команду Revert (возврат) в контекстном меню.

Все резервные копии сохраняются в каталоге __history, вложенном в каталог, в котором находится ваш проект. Этот каталог скрыт, поэтому, чтобы его увидеть, необходимо включить режим отображения скрытых файлов в вашем файловом менеджере. Хотя бы иногда заглядывайте в этот каталог, чтобы его почистить, потому что он имеет тенденцию разрастаться. В моих проектах резервные копии приходится удалять раз в неделю, иначе объем занимаемой ими области диска начинает «зашкаливать» за 10 Мбайт.

Путем настройки можно ограничить количество хранящихся копий. Для этого выберите команду Tools ▸ Options и перейдите в раздел Editor Options, в котором имеется поле File backup limit. По умолчанию создается 10 копий для каждого файла. Если проект состоит из нескольких модулей с небольшим объемом кода, то размер резервных копий будет небольшим. Но если у вас много модулей, да еще с большими изображениями, то каталог __history может быстро разрастись.

Помощь

Справочная система в Delphi 2005 претерпела существенные изменения. В предыдущих версиях использовался старый, но очень удобный формат HLP. Начиная с Delphi 2005, используется формат HXS, для просмотра которого требуется программа Microsoft Document Explorer.

Что представляет собой формат HXS? Это HTML-файлы, собранные в один исполняемый файл. Каждая просматриваемая страница — это HTML-документ. С одной стороны, такая организация файла помощи предоставляет более универсальное и мощное решение, а с другой — к ней необходимо привыкнуть. Однако за два месяца работы мне это так и не удалось, поэтому, когда мне нужна помощь по функциям WinAPI, я обращаюсь к старым файлам помощи из состава Delphi 7.

Запустите файл помощи, выбрав команду Help ▸ Borland Help, и перед вами появится окно справочной системы (рис. 1.14). Вдоль левой стороны окна расположен список разделов помощи. Для поиска информации по ключевому слову перейдите на вкладку Index и введите нужное слово в поле Look for.

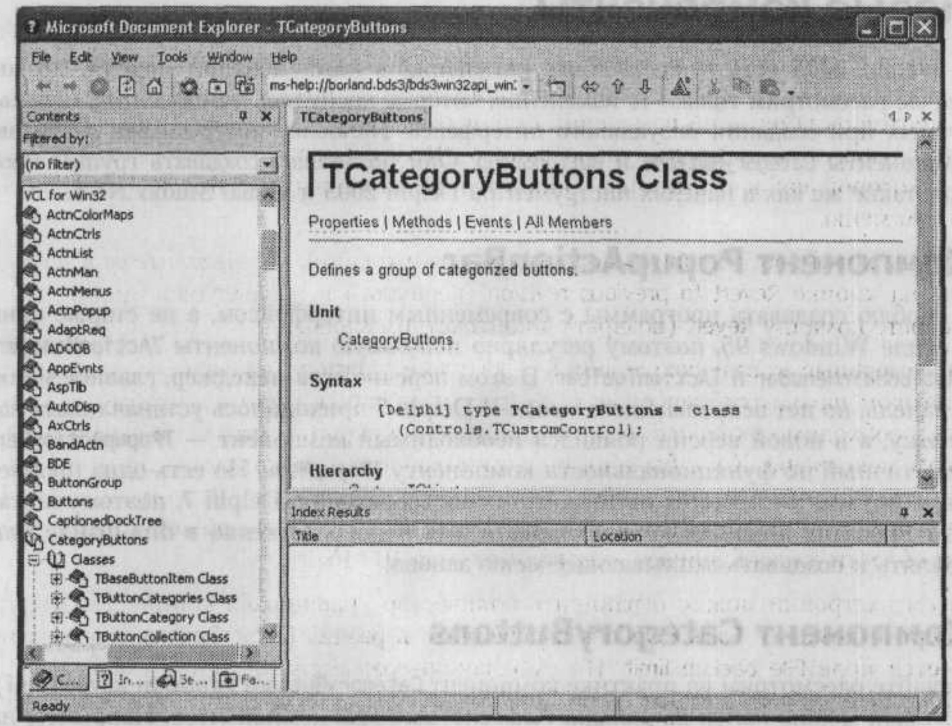


Рис. 1.14. Окно справочной системы в Delphi 2005

Давайте попробуем найти помощь по объекту TCategoryButtons. Введите имя этого объекта. Справа внизу окна будет показан список разделов (колонка Location в списке), в которых найдено ключевое слово. При поиске объектов чаще всего вы будете видеть два раздела:

- Delphi 2005 for .NET Reference — ссылка на версию описания объекта для платформы .NET;
- Delphi 2005 for Win32 Reference — ссылка на описание объекта для платформы Win32.

Описание объекта состоит из следующих частей:

- Заголовок — в заголовке описывается имя и тип найденного ключевого слова/объекта. Под заголовком находятся кнопки для просмотра свойств (Properties), методов (Methods), событий (Events) и всех членов (All Members).
- Unit — модуль, в котором находится описание. Этот модуль необходимо подключить для использования объекта.

- Syntax — синтаксис использования объекта, метода, функции и т. д.
- Hierarchy — иерархия объекта. В этом списке можно увидеть объекты, которые являются родителями для выбранного.

Новые компоненты

В Delphi 2005 есть существенные изменения в компонентной модели, но мы сейчас рассмотрим только те изменения, которые коснулись компонентов, используемых при создании визуального интерфейса. Наиболее интересными я считаю компоненты CategoryButtons и ButtonGroup. Они позволяют создавать группы кнопок, такие же как в панелях инструментов Delphi 2005 и Visual Studio .NET.

Компонент PopupActionBar

Я люблю создавать программы с современным интерфейсом, а не старые окна в стиле Windows 95, поэтому регулярно использую компоненты TActionManager, TActionMainMenuBar и TActionToolbar. В этом перечне есть менеджер, главное меню и панель, но нет всплывающего меню. В Delphi 7 приходилось устанавливать заплатку, а в новой версии появился необходимый компонент — TPopupActionBar, идентичный по функциональности компоненту TPopupMenu. Но есть одна проблема — его имя отличается от того, что использовалось в Delphi 7, поэтому в старых проектах придется корректировать имя непосредственно в dfm-файле или удалять и создавать всплывающее меню заново.

Компонент CategoryButtons

Давайте рассмотрим на практике компонент CategoryButtons (кнопки категорий), который можно найти на вкладке Additional палитры компонентов. Например, на рис. 1.15. компонент располагается вдоль левой стороны окна. Давайте воспроизведем этот пример и познакомимся с проблемами, которые могут возникнуть при работе с компонентом CategoryButtons. А проблемы есть.

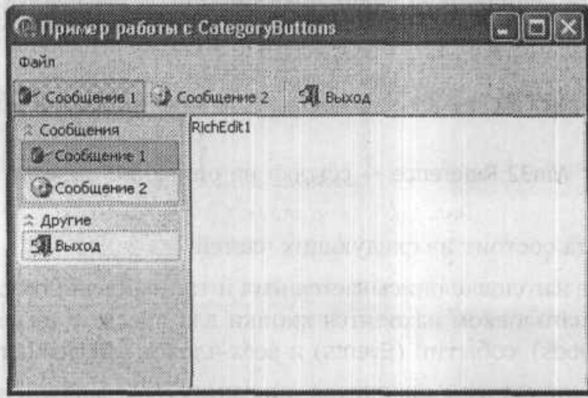


Рис. 1.15. Пример компонента TCategoryButtons

Итак, для создания этого компонента поместим на форму следующие компоненты:

- TImageList (вкладка Win32) — в этом компоненте хранятся изображения для кнопок;
- TActionManager (вкладка Additional) — компонент для создания и управления действиями;
- TActionMainMenuBar (вкладка Additional) — компонент для создания главного меню;
- TActionToolbar (вкладка Additional) — компонент для создания панели задач;
- TCategoryButtons (вкладка Additional) — компонент для создания группы кнопок.

В компоненте TActionManager создайте три действия с именами «Сообщение 1», «Сообщение 2», «Выход» и для первых двух действий напишите обработчики события OnExecute, в которых просто выводится на экран сообщение:

```
Application.MessageBox('Нажата кнопка "Сообщение 1"',
'Внимание', MB_OK);
```

Разместите действия на панели задач и меню простым перетаскиванием. Чтобы действия появились в компоненте TCategoryButtons, нужно затратить чуть больше усилий.

1. Дважды щелкните на компоненте, и перед вами откроется окно редактора категорий (рис. 1.16).
2. Нажмите клавишу Insert, чтобы вставить новую категорию.
3. В объектном инспекторе задайте имя категории в свойстве Caption.
4. Дважды щелкните на свойстве Items, чтобы открыть окно редактора элементов выбранной категории. Окно имеет такой же вид, как и окно редактора категорий.
5. Для создания нового элемента также нажмите клавишу Insert и в свойстве Action выберите нужное действие.

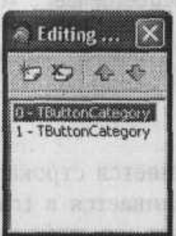


Рис. 1.16. Окно редактора категорий

Таким образом, мы создали две категории. В первой категории находятся две кнопки вывода сообщений, а во второй категории кнопка для выхода из программы.

Чтобы кнопки выглядели красивее, можно выделить компонент CategoryButtons и выполнить следующие действия.

1. В свойстве Images укажите компонент TImageList с изображениями. Связь между компонентами не образуется автоматически, ее приходится задавать вручную, иначе на кнопках не будет изображений.
2. В разделе ButtonOptions можно изменить следующие два свойства:
   - boFullSize — установите в true, чтобы кнопки отображались полностью (изображение вместе с подписью);
   - boVerticalCategory — установите в false, чтобы имена категорий прорисовывались горизонтально (при вертикальной прорисовке имена не всегда удобно читать).

Запустите программу. Попробуйте щелкнуть на одной из кнопок в компоненте CategoryButtons. Обратите внимание, что после появления окна сообщения, которое мы прописали в обработчике событий, кнопка остается нажатой. При этом на панели инструментов и в меню кнопка не выглядит нажатой, хотя выполняется одно и то же действие TAction. Такая несогласованность действий вряд ли понравится пользователю программы, поэтому лучше будет исправить ошибку, правда, для этого придется добавить несколько строк кода и изменить парочку свойств.

Сначала установите в свойстве GroupIndex всех кнопок, которые отображаются в компоненте CategoryButtons, одинаковое число (это необходимо делать в ActionManager в свойстве GroupIndex каждого действия). Кнопки с одинаковым числом в свойстве GroupIndex работают как группа. Если одна из них оказывается нажатой, то все остальные автоматически возвращаются в исходное состояние.

Таким образом, можно создать несколько групп, которые будут работать схожим образом, например в текстовом редакторе одна группа отвечает за выравнивание (кнопки **Влево**, **Вправо**, **По центру**), другая — за вид документа (кнопки **Обычный**, **Веб-документ**, **Разметка**) и т. д.

Добавим в код обработчика OnExecute изменения, отражающие состояния кнопки. Например, для кнопки «Сообщение 1» код обработчика будет следующим:

```
procedure TForm1.acMessage1Execute(Sender: TObject);
begin
 Application.MessageBox('Нажата кнопка "Сообщение 1"',
   'Внимание', MB_OK);
 acMessage1.Checked:=true;
end;
```

После отображения окна выполняется строка кода, в которой свойство Checked выполненного действия устанавливается в true. Все остальные кнопки из этой группы автоматически изменят это же свойство на false.

Вот таким простым приемом мы исправили недостаток в работе компонента CategoryButtons. Если у вас не дублируются кнопки панелей категорий и инструментов (или меню), то этот трюк не нужен.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch01\CategoryButtons.

Компонент ButtonGroup

По своей сути компонент ButtonGroup похож на CategoryButtons, даже выглядят они почти одинаково (рис. 1.17), но категории не поддерживаются. Для создания группы компонентов установите на форму компонент ButtonGroup с вкладки Additional. Если вы хотите видеть на кнопках изображения, свяжите компонент ButtonGroup с компонентом ImageList вручную.

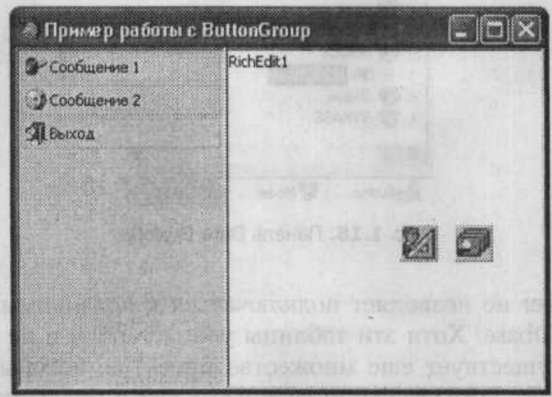


Рис. 1.17. Группы кнопок

Для добавления кнопок в группу дважды щелкните на компоненте ButtonGroup и перед вами откроется окно редактирования списка кнопок. Для добавления новой кнопки нажмите клавишу Insert.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch01\ButtonGroup.

Базы данных

Что еще можно придумать для управления базами данных, учитывая, что эта тема уже проработана «вдоль и поперек»? С помощью Delphi всегда было намного проще создавать приложения баз данных, но программисты Borland нашли способ сделать нашу жизнь еще проще. Куда уж проще? Оказывается, есть куда.

В Delphi 2005 появилась панель, с помощью которой можно управлять базой данных и просматривать данные прямо из среды разработки. К тому же очень удобно всегда держать перед глазами структуру базы данных, видеть имена таблиц и полей. Это позволяет минимизировать вероятность возникновения ошибок при создании SQL-запросов. Теперь нет необходимости постоянно «прыгать» между окнами приложений или держать перед глазами распечатанную структуру базы данных.

На правой верхней панели, где располагается менеджер проектов, есть несколько вкладок. Одна из них — Data Explorer (проводник данных) — показана на рис. 1.18. На ней уже есть заготовки для подключения к основным базам данных DB2, Interbase, MSAccess, MSSQL, Oracle и SYBASE.

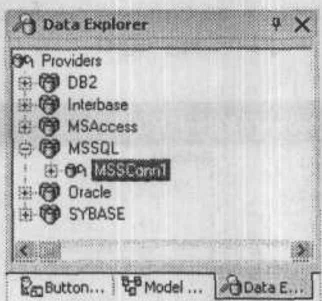


Рис. 1.18. Панель Data Explorer

Панель Data Explorer не позволяет подключаться к локальным таблицам, таким как Paradox или Dbase. Хотя эти таблицы уже устарели и не рекомендуются к использованию, существует еще множество проектов, которые приходится сопровождать. Формат DBF особенно распространен в государственных структурах (пенсионные и медицинские фонды, налоговая инспекция). Для решения этой проблемы, можно связать DBF-таблицу с Access (подключить ее как внешний источник данных), а из Delphi подключиться к файлу Access.

Давайте рассмотрим пример работы с базой данных MS SQL Server. Раскройте ветвь MSSQL иерархического списка на панели Data Explorer, и перед вами откроется один элемент соединения. Для создания нового соединения щелкните правой кнопкой мыши на ветви MSSQL и в контекстном меню выберите команду Add New Connection (добавить новое соединение). Перед вами откроется окно, в котором необходимо выбрать провайдер (тип базы данных) и указать имя соединения, которое будет отображаться в иерархическом списке панели Data Explorer.

Теперь у нас в списке есть новый пункт соединения, но его параметры установлены по умолчанию. Среда разработки Delphi еще «не знает», к какому серверу мы хотим подключаться, какая база данных будет использоваться, каковы имя и *пароль* подключения, без которых SQL Server нас просто не пустит к таблицам. Чтобы все это указать, необходимо отредактировать новое соединение.

Для редактирования существующего соединения щелкните правой кнопкой на его имени (например, MSSConn1) и в контекстном меню выберите команду Modify Connection. Перед вами откроется окно Connections Editor (рис. 1.19). Для SQL Server в этом окне необходимо задать следующие параметры:

- Database — база данных, к которой необходимо подключиться;
- Hostname — имя сервера SQL Server;
- OSAutentication — если в этом поле указать значение `true`, то при подключении будут использоваться текущие имя и пароль, с которыми вы вошли в компьютер, а чтобы задать другие параметры, укажите в этом поле значение `false`;

- Password и Username — пароль и имя пользователя, используемые для авторизации на сервере.

Теперь можно раскрыть ветвь соединения и просмотреть таблицы в базе данных и их поля. Если щелкнуть правой кнопкой мыши на пункте Table, то в контекстном меню можно увидеть команду New Table, служащую для создания таблицы прямо из среды разработки Delphi 2005. В контекстном меню таблиц есть также команды для просмотра данных, редактирования структуры таблицы и удаления.

Рис. 1.19. Окно редактирования соединения с базой данных

Команды редактирования данных в таблицах актуальны для баз данных, в поставку которых не входят средства для манипуляции данными.

Синхронное редактирование

Возможность синхронного редактирования можно отнести к нововведениям редактора кода, но я решил поговорить о ней в отдельном разделе, потому что эта возможность действительно удобна и иногда незаменима. Допустим, у вас есть процедура, которая содержит следующий код:

```
Procedure Calculate();
var
 i, index :Integer;
begin
 index:=0;
 for i:=0 to 100 do
 index:=index+i;
 ShowMessage(IntToStr(Index));
end;
```

Допустим, нужно переименовать переменную index в этой процедуре. Для этого можно внести пять изменений вручную или воспользоваться механизмом поиска и замены, что не всегда дает необходимый результат. Например, если в коде будет еще и переменная index1, то во время поиска и замены будет переименована и эта переменная. Конечно же, вы можете при поиске ограничиться только целыми словами, но это — частный случай решения проблемы.

В Delphi 2005 появился более интеллектуальный метод редактирования одинаковых имен. Выделите код процедуры. Слева от кода в окне редактора рядом с номерами строк появится кнопка Sync Edit Mode (режим синхронного редактирования). Щелкнув на этой кнопке, мы перейдем в режим синхронного редактирования (рис. 1.20).

```
23  {$R *.dfm}
24
25  procedure TForm1.FormCreate(Sender: TObject);
    var
27   i, index :Integer;
28  begin
29   index:=0;
30   for i:=0 to 100 do
31    index:=index+i;
32   ShowMessage(IntToStr(index));
33  end;
34
35  end.
36
```

Рис. 1.20. Режим синхронного редактирования

При переходе в режим синхронного редактирования редактор находит все одинаковые слова в выделенном фрагменте кода и обводит их синим квадратным контуром. Выделенная переменная, имеющая повторения, просто подчеркивается синей линией (на рис. 1.20 выделена переменная i).

Попробуйте теперь в этом режиме изменить переменную index. Одновременно с ней изменят свое имя все найденные переменные в выделенном фрагменте с таким же именем.

Использование режима синхронного редактирования для изменения имен — более интеллектуальный способ, чем поиск и замена, когда нужно отредактировать целое слово. Но если нужно заменить несколько слов (например, конструкцию типа Button.Text заменить конструкцией Label.Caption), метод поиска и замены может оказаться удобней. Выбирайте метод, который больше подходит именно вам и лучше решает поставленную задачу.

Поиск метода

В Delphi 2005 мне понравилась возможность перехода от метода к методу. Например, пусть у вас есть следующий код:

```
procedure ProcName;
begin
```

```
    SomeProc;
  end;
  ...
  procedure SomeProc;
  begin
  ...
  end;
```

В процедуре ProcName вызывается процедура SomeProc. Допустим, между этими процедурами большой объем кода. Как быстро перейти из кода ProcName в SomeProc? Достаточно, удерживая клавишу Crtl, щелкнуть на имени процедуры SomeProc. В результате вы окажитесь в теле SomeProc.

Поиск компонента

Новая палитра компонентов немного раздражает. Чтобы найти нужный компонент, иногда приходится тратить очень много времени, особенно если у вас установлен не один десяток компонентов сторонних разработчиков.

Однако если вы хотя бы приблизительно знаете имя нужного компонента, то в Delphi 2005 найти его несложно. Активизируйте палитру компонентов и убедитесь, что кнопка Filter or unfilter the current items нажата, или нажмите ее щелчком мыши. После этого начните набирать имя искомого компонента; вместо палитры появится список компонентов, имена которых начинаются с введенного слова. Чем больше букв искомого имени вы введете, тем меньше будет список.

Чтобы снова отобразить всю палитру компонентов, повторным щелчком отожмите кнопку Filter or unfilter the current items.

Когда вы находитесь в окне редактора кода, в палитре компонентов отображаются заготовки кода (Code Snippets) и кнопки создания и добавления в проект окон и модулей. Их также можно искать, если нажата кнопка Filter or unfilter the current items.

Управление файлами

Бывают случаи, когда некоторые файлы проекта необходимо переименовать. Если подобный файл является модулем с формой, то переименование приводит к тяжелым последствиям, поскольку заставляет корректировать слишком много ссылок. В предыдущих версиях Delphi было проще удалить файл, переименовать его в файловом менеджере и снова добавить. После этого достаточно было только изменить ссылки на модуль в разделе uses всех модулей, ссылающихся на переименованный файл.

В Delphi 2005 переименование происходит намного проще. Щелкните на имени файла в менеджере проекта правой кнопкой мыши, в контекстном меню выберите команду Rename и введите новое имя. После этого все готово к продолжению работы.

Глава 2

Советы и секреты

В этой главе мы будем рассматривать интересные алгоритмы работы с компонентами и объектами, которые предоставляет нам Delphi и которые вы можете использовать в своих проектах.

Большая часть главы посвящена практическим советам и описанию решений различных типовых задач, с которыми вы можете встретиться в повседневной жизни. Здесь представлены наиболее часто используемые приемы программирования, применяемые для разрешения проблем, с которыми автору приходится иметь дело чуть ли не каждый день. Возможно, вы уже тоже сталкивались с аналогичными проблемами или столкнетесь с ними в будущем.

Мы познакомимся с некоторыми секретами, которые позволят вам добиться максимальных результатов при работе с Delphi. Часть рассматриваемых примеров относится к Delphi 2005, другие работают в Delphi 7, но большинство остаются работоспособными в любой версии.

Я всегда стараюсь писать код, который не привязан к определенной версии Delphi, чтобы не приходилось вносить много изменений при обновлении среды разработки. (Обновление Delphi происходит достаточно часто, а регулярные внесения изменений в код только усложняют жизнь разработчику.) Советую и вам следовать этому правилу и по возможности не привязываться к версиям.

Секреты DFM

Каждый модуль, который содержит визуальную форму, состоит из двух файлов со следующими расширениями:

- .pas — исходный код модуля;
- .dfm — описание визуального содержимого формы (объекты, их свойства и расположение).

Мы чаще всего редактируем pas-файл, а про dfm-файл некоторые программисты даже не знают или просто не обращают на него внимания. Но иногда этот файл действительно необходим, и желательно разбираться в его формате и структуре. На самом деле формат dfm-файла достаточно прост. Его составляют простые текстовые команды, которые можно редактировать в любом текстовом редакторе. Если нужно что-то подправить, я открываю dfm-файл в блокноте и редактирую нужные параметры вручную.

Например, у меня был проект, в котором находилось 40 компонентов TTable. Каждый из них был активен и настроен на определенную базу данных. Однажды я переименовал базу данных и попытался открыть проект. Открытие происходило очень долго, потому что дизайнер форм запрашивал для каждого компонента соединение с базой данных, и если его не было в течение определенного периода времени (TimeOut), то выдавалось сообщение об ошибке. Если бы тайм-аут был бесконечным, то форма с компонентами вообще никогда бы не открылась.

Проблема решается очень просто. Необходимо просто вручную отключить соединение с базой, а это делается редактированием в текстовом редакторе (например, в блокноте) dfm-файла.

В листинге 2.1 я показал пример простого dfm-файла, в котором описана форма, содержащая два компонента — TRichEdit (поле ввода) и TButton (кнопка). Давайте рассмотрим формат файла на примере этого листинга.

Листинг 2.1. Содержимое dfm-файла

```
object Form1: TForm1
  Left = 0
  Top = 0
  Width = 394
  Height = 284
  Caption = #1055#1088#1080#1084#1077#1088' '#1089' Example'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object RichEdit1: TRichEdit
    Left = 153
    Top = 0
    Width = 233
    Height = 250
    Align = alClient
    Lines.Strings = (
      'RichEdit1')
    TabOrder = 1
  end
  object Button1: TButton
    Left = 288
```

продолжение

Листинг 2.1 (продолжение)

```
    Top = 224
    Width = 75
    Height = 25
    Caption = 'OK'
    ModalResult = 1
    TabOrder = 0
  end
end
```

Первая строка этого dfm-файла, начинающая описание объекта, выглядит так:

```
object Form1: TForm1
```

Ключевое слово `object` говорит о том, что сейчас будет описываться объект. После него должно стоять имя объекта, затем двоеточие и тип объекта. Таким образом, если какая-то форма не открывается из-за отсутствия компонента, этот компонент можно удалить или изменить его тип на совместимый с ним. Предположим, вы установили более совершенную сетку `DBGrid` под названием `TExDBGrid`, но потом компонент затерялся или вы не смогли его перенести на новую версию Delphi, а проект из-за этого не открывается должным образом. Для решения проблемы можно изменить тип объекта на стандартный `TDBGrid`, и проект откроется. Но при этом могут появляться сообщения об ошибках, извещающие о том, что какое-то свойство, специфичное для `TExDBGrid`, не найдено и будет удалено. Но это уже не так страшно, так как проект все равно открывается.

После строки с ключевым словом `object` идет описание объекта, которое заканчивается, когда анализатор встречает слово `end`. Таким образом, полный вид описания объекта выглядит следующим образом:

```
object имя : тип
  Описание
end
```

Никаких точек с запятыми быть не должно.

Объекты могут быть вложенными. Например, в следующем примере кнопка вложена в форму, то есть кнопка в дизайнере располагается на форме:

```
object Form1 : TForm
 Описание
 object Button1 : TButton
  Описание
 end
end
```

Как видите, ничего сложного тут нет. Неудобно? Да. С помощью дизайнера создавать формы намного проще.

Теперь посмотрим на описание. Оно еще проще, потому что содержит всего лишь строки в виде:

```
Свойство = Значение
```

В следующем примере в описании только одна строка, которая задает левую позицию кнопки объекта:

```
object Button1 : TButton
  Left = 0
```

```
end
```

Все остальные значения по умолчанию задаются конструктором этого объекта. Таким образом, если какому-то свойству нужно присвоить значение по умолчанию, достаточно удалить его описание в dfm-файле.

Вы можете создавать и изменять свойства вручную. Имена свойств (их нужно писать слева) представляют собой те же имена, которые вы видите в объектном инспекторе или в разделе Properties файла помощи по объекту.

Значение зависит от типа. Если свойство числовое, то ему можно присвоить число. А вот со строками немного сложнее, потому что русский язык записывается в кодировке Unicode, чтобы проект был совместим с Kylix (ОС Linux). Английский текст пишется в простых одинарных кавычках, а русский — вне кавычек и с кодированием. Например:

```
'FROM ['#1041#1072#1079#1072#1058#1077#1083#1077#1092']'
```

Здесь, в самом начале идет английский текст в чистом виде ('FROM ['), затем записано закодированное слово на русском языке, а в самом конце строки в кавычках указан символ закрывающей квадратной скобки (']').

Теперь посмотрим, как можно решить проблему открытия модуля, в котором компоненты ADO связаны с несуществующей базой данных. Если вы использовали компонент ADOConnection, то сначала ищем его объявление:

```
object ADOConnection1: TADOConnection
  Connected = True
  ConnectionString =
    'Provider=Microsoft.Jet.OLEDB.4.0;Data Source=1.mdb'
  LoginPrompt = False
  Mode = cmShareDenyNone
  Provider = 'Microsoft.Jet.OLEDB.4.0'
  Left = 32
  Top = 16
end
```

В первой строке описания свойству Connected присваивается значение True. Просто изменяем его на False в любом текстовом редакторе, и готово. После этого ищем все описания объектов TADOTable и TADOQuery и у них вручную присваиваем свойству Active значение False.

После этого среда разработки Delphi сможет открыть проект, не пытаясь соединиться с базой данных.

Расширение возможностей стандартных компонентов

Очень часто нам необходимо расширить те или иные возможности компонентов. Я уже не раз встречался с такой проблемой. Самым правильным способом ее решения является создание нового компонента-потомка, реализующего необходимые действия.

Однако иногда можно и даже нужно отступить от этого правила. Например, пусть у вас в проекте используются 100 кнопок TButton, и вы решаете наделить

эти кнопки новым свойством. Если следовать классическому методу решения проблемы, то потребуется сначала создать новый компонент, а потом поступить одним из двух способов.

- Вручную изменить в dfm- и pas-файлах предка для каждого компонента. Когда их 100 штук, процесс изменения может затянуться и будет очень утомительным.
- Заново создать все кнопки. Старые нужно удалить и создать новые кнопки от нового компонента. Этот способ требует еще больше времени, чем первый, особенно если вы не используете действия (TAction) и придется переделывать каждое свойство и каждый обработчик события кнопки.

Гораздо проще изменить стандартный компонент TButton и использовать его. Вариантов изменения может быть два — глобальное изменение и локальное. В любом случае изменения производятся одинаково, разница только в том, как подключать файл.

У нас есть возможность изменения стандартных компонентов благодаря тому, что все исходные коды компонентов входят в поставку Delphi. Однако самое странное, что этих исходных кодов в Delphi 2005 я не нашел. Возможно, это связано с тем, что у меня бета-версия, а в основной версии исходные коды появятся, потому что они действительно необходимы.

В Delphi 7 исходные коды VCL-компонентов находятся в каталоге Delphi7/Sources/Vcl. Объявление кнопки TButton находится в модуле StdCtrls.pas. Давайте откроем этот файл и отредактируем объект TButton. Я специально выбрал именно этот объект, потому что при локальной компиляции с ним возникает множество проблем.

Итак, найдите описание компонента TButton и добавьте в раздел public новую процедуру:

```
procedure ShowButtonMessage(sTest:String);
```

Нам не нужны сложные манипуляции, главное — добиться изменения работы стандартного компонента. Нажимаем клавиши Ctrl+Shift+C, чтобы среда Delphi создала шаблон для процедуры. В ней вызовем окно с сообщением, которое передается в качестве параметра:

```
procedure TButtonControl.ShowButtonMessage(
  sTest: String);
begin
 MessageBox(0, PChar(sTest), 'Тест', 0);
end;
```

Теперь необходимо, чтобы при компиляции среда Delphi использовала модифицированную версию файла. Сначала рассмотрим глобальный метод. Выберите команду Tools ▸ Environment Options. Перед вами откроется окно настройки среды Delphi. Перейдите на вкладку Library. Здесь в поле Library path перечислены пути, по которым компилятор должен искать модули. Щелкните на кнопке с тремя точками, чтобы было удобнее смотреть и редактировать список путей в специальном окне (рис. 2.1).

Обратите внимание на первую строку в списке — там указан путь $(DELPHI)\Lib. Эта строка означает, что сначала компилятор будет искать модули в каталоге

\lib каталога установки Delphi. Если нужный dcu- или pas-файл в этом каталоге найти не удастся, то будут просматриваться остальные каталоги в списке.

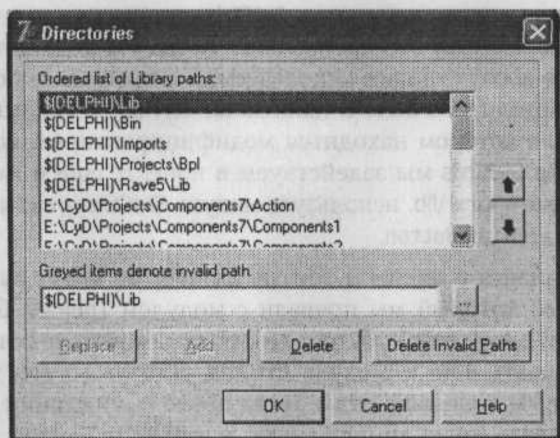


Рис. 2.1. Окно настройки каталогов для поиска модулей

Наша задача добавить путь к каталогу с исходными кодами VCL-модулей и сделать этот путь первым в списке, чтобы компилятор использовал именно его. Для этого выполняем следующие действия.

1. В поле ввода под списком каталогов введите путь $(DELPHI)\Source\Vcl.
2. Щелкните на кнопке Add, чтобы добавить путь.
3. Выделите добавленный путь и, щелкая на кнопке с направленной вверх стрелкой, переместите его на первую строку списка.
4. Сохраните изменения щелчком на кнопке OK.

Теперь можно компилировать проект.

Таким образом, мы глобально изменили параметры, заставив компилироваться все модули из каталога VCL. Однако это не очень хорошо, ведь мы изменили только один файл — StdCtrls.pas.

Для локального подключения файла StdCtrls.pas его необходимо скопировать в тот же каталог, в котором вы храните исходный код проекта. В этом случае никаких параметров Delphi изменять не надо, потому что каталог проекта всегда проверяется первым. Найдя в нем файл StdCtrls.pas, компилятор уже не будет проверять каталог \lib, а использует модифицированную версию компонента TButton.

Попробуйте выполнить локальную компиляцию. Вот тут-то и возникают проблемы. Сначала компилятор укажет на модуль Themes и сообщит, что модуль Dialogs скомпилирован с другой версией компонента TButton. В данном случае проблема решается просто. Обратите внимание, что модуль Themes подключен в секции uses раздела implementation. Если перенести подключение в секцию uses раздела interface (в начале модуля), то ошибка будет исправлена. Такой трюк

работает не всегда и является скорее исключением из правил. Далее мы рассмотрим более надежный вариант решения проблемы.

Снова попробуйте скомпилировать программу, и снова произойдет ошибка. Вы получите сообщение о том, что модуль ExtCtrls скомпилирован с другой версией компонента TButton. Почему это произошло? Когда мы подключали измененный файл глобально, то абсолютно все используемые в программе модули из каталога \Source\Vcl перекомпилировались. В данном же случае перекомпилируется только модуль StdCtrls, в котором находится модифицированный компонент TButton. Но помимо модуля StdCtrls мы задействуем в проекте еще и модуль ExtCtrls, который берется из каталога \lib, использует модуль StdCtrls и скомпилирован с другой версией компонента TButton.

Два разных компонента с одним и тем же именем не могут существовать в одном проекте. Трюк, который мы провели с модулем Themes, больше не проходит. Тут нужно другое решение. А оно лежит на самой поверхности — достаточно перекомпилировать еще и модуль ExtCtrls. Для этого его тоже скопируйте в каталог с исходным кодом проекта. Тогда после компиляции этот модуль также будет использовать модифицированный компонент TButton.

Чтобы не было проблем с компиляцией, скопируйте в каталог с исходным кодом проекта файлы Dialogs.pas и Buttons.pas.

Вот таким нехитрым способом несколько лет назад я решил одну очень серьезную проблему. В одной из версий Delphi (точно не помню, но, кажется, это была Delphi 4) в сетке StringGrid один из методов работал не совсем так, как мне бы хотелось. Я модифицировал его и подключил к программе локально.

Лучше использовать локальное подключение, потому что далеко не всегда модифицированная версия компонента может понадобиться вам во всех ваших проектах. Да и компилировать все стандартные модули из исходных кодов нет смысла, когда уже есть скомпилированные версии в каталоге \lib.

ПРИМЕЧАНИЕ

> Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\Vcl.

Доступ к защищенным свойствам и методам

Я уже не раз замечал, что разработчики Delphi иногда прячут в секции private или protected очень полезные свойства и методы компонентов. Я понимаю, что они запрещают доступ к тем свойствам и методам, неверное обращение с которыми может привести к краху программы. И все же, иногда доступ к «запретному плоду» просто необходим.

Как получить доступ к защищенной переменной или методу? Рассмотрим проблему на примере кнопки. У компонента TButton есть метод SetButtonStyle, объявленный в секции protected. Как обмануть Delphi и получить доступ к этому

методу напрямую? Если написать в программе следующий код, то во время компиляции произойдет ошибка:

```
Button1.SetButtonStyle(true);
```

У компилятора Delphi есть один недостаток (а может быть, достоинство, сказать трудно). Если два объекта объявлены в одном модуле, то все их свойства и методы доступны друг другу. При этом не имеет значения, где именно в модуле объявлены эти методы и свойства, а также защищены они или нет. Но ведь кнопка и наш проект объявлены в разных модулях, поэтому эта возможность на первый взгляд кажется бесполезной. Не торопитесь. У объектно-ориентированного программирования есть еще одна очень полезная возможность — наследование. При наследовании потомку может быть назначен любой предок, и код будет работать верно.

Давайте объединим эти две возможности, чтобы получить доступ к защищенному методу. Итак, в модуле нашей формы объявляем новый класс, который будет происходить от TButton. Объявление выглядит следующим образом (оно должно быть в разделе type):

```
TMyButton = class(TButton)
end;
```

Теперь в любом месте нашего модуля можно использовать защищенные методы кнопки. Допустим, что у нас на форме есть кнопка Button1 типа TButton. Этот объект является предком для TMyButton, поэтому запись TMyButton(Button1) является вполне корректной. А так как класс TMyButton объявлен в том же модуле, то легко можно получить доступ к его защищенным методам и свойствам. Для этого достаточно написать код:

```
TMyButton(Button1).SetButtonStyle(true);
```

Получается, что когда мы объявили наследника от кнопки, все его методы переместились в наш модуль. А так как в одном модуле все классы являются дружественными и «видят» все защищенные свойства и методы друг друга, мы с легкостью смогли их прочитать.

Этот способ хорош только тогда, когда вам нужно получить доступ к защищенному (из раздела protected) свойству или методу. Что же касается закрытых свойств и методов (объявленных в разделе private), все они недоступны потомкам, а значит, не переносятся в наш модуль.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\AccessToPrivate.

Сохранение данных проекта

Проблема сохранения данных своей программы в файле встает перед большинством программистов. Например, пусть вы разрабатываете какой-либо визуальный редактор и нужно сохранить модель в файле для того, чтобы иметь возможность впоследствии ее загрузить. Почему-то это вызывает достаточно серьезные

проблемы у некоторых программистов. Давайте рассмотрим один из методов решения этой задачи.

Я уже не раз говорил в своих работах, что по возможности нужно использовать объекты. Почему? Если вы уже имеете опыт программирования и использовали функции WinAPI для работы с файлами, то могли заметить, что существует несколько разновидностей этих функций. Например, для открытия файла можно задействовать функцию _lopen, OpenFile или CreateFile.

Почему так много функций в библиотеке WinAPI, выполняющих одни и те же действия? В первых версиях Windows была только функция — _lopen. Однако возможности этой функции слишком ограниченны, она поддерживает только несколько флагов, влияющих на режим открытия файла, и не имеет никаких средств обеспечения безопасности.

С переходом на 32-битные версии Windows и FAT32 понадобилась более мощная функция, которая могла бы работать с файлами большого размера и поддерживать раздельное чтение (когда два разных процесса получают доступ к одному файлу). Такой функцией стала функция FileOpen, а прежняя функция _lopen осталась для совместимости со старым программным обеспечением.

Со временем оказалось, что у функции FileOpen также не хватает возможностей, и в WinAPI снова ввели новую функцию — CreateFile. Мир информационных технологий не стоит на месте, а постоянно развивается, поэтому функции будут появляться снова и снова. А чтобы переходить на эти новые функции, без использования объектов пришлось бы переписывать код обработки файлов во всех своих проектах.

Проблема решается просто — нужно применить объект, который умеет работать с файлами. При появлении новых возможностей будет достаточно изменить только этот объект, а все проекты просто перекомпилировать с новой версией объекта без внесения в них каких бы то ни было изменений. Таким образом, объекты позволяют экономить очень много времени и беречь нервы.

Еще одно достоинство объектов — они не привязаны к конкретной платформе. Например, если работу с файлами в проектах организовать на основе все тех же функций WinAPI для платформы Win32, то при переносе проекта на платформу .NET весь код придется переписывать.

Для работы с файлами в Delphi есть объект TFileStream, который представляет собой поток данных и является потомком объекта TStream. Если вам требуется обработка файлов, используйте этот объект.

Итак, давайте рассмотрим практический пример сохранения данных и напишем код, в котором реализуем следующие возможности:

- на форме в случайной точке должен появляться компонент TImage;
- в созданный компонент должно загружаться случайное изображение из набора TImageList;
- программа должна иметь возможность сохранять положения компонентов и изображение в файле (сохранять необходимо именно изображение, а не его индекс в списке);

■ в программе должна быть реализована возможность загрузки данных из файла.

Для реализации проекта я поместил на форму меню и панель с кнопками Создать, Открыть, Сохранить и Добавить картинку (рис. 2.2). Помимо этого на форме нужно разместить компоненты OpenDialog и SaveDialog, предназначенные для отображения стандартных окон открытия/сохранения файлов, и список ImageList с изображениями. Для примера я поместил в список 6 изображений размером 32×32.

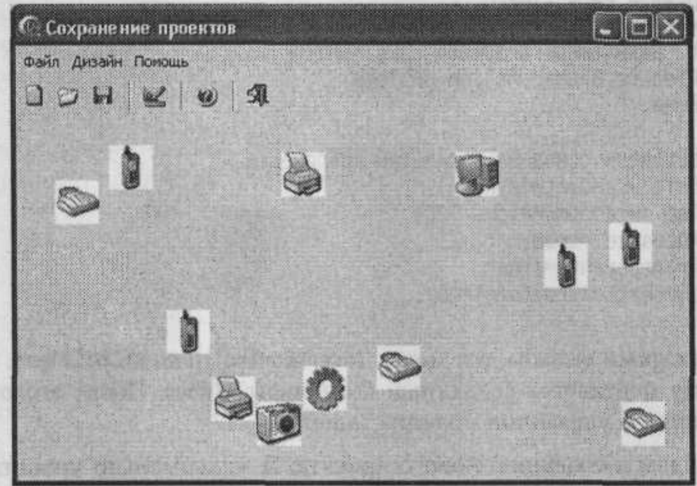


Рис. 2.2. Форма будущей программы

Для упрощения задачи в центре окна на всю клиентскую область «растянут» компонент TPanel. Именно на нем будут создаваться изображения. Впоследствии вы увидите, как это будет происходить.

Теперь рассмотрим реализацию всего вышесказанного. Начнем с создания компонентов на форме. Я всегда начинаю с этого, чтобы потом четко видеть, где и что хранится, как это можно сохранить и затем загрузить. Итак, по щелчку на кнопке Добавить картинку будет выполняться код, представленный в листинге 2.2. Я постарался сделать код как можно проще для чтения, чтобы с ним можно было быстро разобраться. Ведь наша основная задача в данном примере — обработка файлов, а не создание компонентов.

Давайте подробнее рассмотрим код. Сначала создается компонент TImage. Во время его создания методу Create передается указатель на панель, на которую мы поместим наше изображение. Затем выбирается случайное изображение из списка ImageList2 и помещается во временный объект Bitmap. Этот объект далее копируется в созданный компонент TImage.

В качестве дополнения мы случайным образом устанавливаем позицию компонента и изменяем его свойства Autosize и Transparent на true, чтобы загруженное изображение было прозрачным и компонент автоматически принял его размеры.

Листинг 2.2. Код создания компонента TImage на форме

```
procedure TSaveProjectForm.acAddImageExecute(
  Sender: TObject);
var
 imgTemp:TImage;
 bmpTemp:TBitmap;
begin
 imgTemp:=TImage.Create(plClient);

 // Получаем случайную картинку из ImageList
 // и загружаем ее в TImage
 bmpTemp:=TBitmap.Create;
 ImageList2.GetBitmap(random(6), bmpTemp);
 imgTemp.Picture.Bitmap.Assign(bmpTemp);
 bmpTemp.Free;

 // Устанавливаем основные параметры компонента
 imgTemp.Left:=random(Width-20);
 imgTemp.Top :=random(Height-100);
 imgTemp.Autosize:=true;
 imgTemp.Transparent:=true;
 plClient.InsertControl(imgTemp);
end;
```

В последней строке вызывается метод `InsertControl` панели `plClient`. В качестве параметра ему передается созданный компонент `TImage`. После этого изображение размещается в указанной позиции панели.

Теперь перейдем к созданию нового проекта. Я максимально упростил пример, и список создаваемых компонентов `TImage` нигде не хранится, поэтому достаточно просто уничтожить все изображения, после чего можно считать, что новый проект создан. Итак, по щелчку на кнопке **Создать** у нас будут выполняться всего две строки кода:

```
procedure TSaveProjectForm.acNewProjectExecute(
    Sender: TObject);
begin
 plClient.DestroyComponents();
 FileName:='';
end;
```

В первой строке вызывается метод `DestroyComponents` панели, на которой могут находиться компоненты `TImage`. Так как все наши изображения находятся на панели, они уничтожаются. Во второй строке обнуляется строковая переменная `FileName`. Это может быть глобальная переменная, в которой сохраняется имя файла текущего проекта. Если имя пустое, то будет вызываться стандартное окно для выбора файла.

Вот теперь переходим к процессу сохранения данных. По щелчку на кнопке **Сохранить** будет выполняться код из листинга 2.3.

Листинг 2.3. Сохранение проекта

```
procedure TSaveProjectForm.acSaveExecute(Sender: TObject);
var
 fs:TFileStream;
```

```
  ms:TMemoryStream;
  i, index:Integer;
begin
  //Если имя файла не задано, то вызвать "Сохранить как"
  if FileName='' then
   begin
    acSaveAsExecute(nil);
    exit;
   end;

  //Сохраняем проект
  fs:=TFileStream.Create(FileName, fmCreate);
  try
   //Записываем сигнатуру
   fs.Write('proj', 4);

   //Записываем количество элементов
   index:=plClient.ComponentCount;
   fs.Write(index, sizeof(index));

   //Записываем информацию о картинках
   for i:=0 to plClient.ComponentCount-1 do
    begin
    //Левая позиция
    fs.Write(TImage(
       plClient.Components[i]).Left, sizeof(index));

     //Верхняя позиция
     fs.Write(TImage(
       plClient.Components[i]).Top, sizeof(index));

     //Картинку загружаем в память
     ms:=TMemoryStream.Create;
     TImage(plClient.Components[i]).Picture.Bitmap.SaveToStream(ms);
     ms.Seek(0, soFromBeginning);
     //Сохраняем размер картинки
     index:=ms.Size;
     fs.Write(index, sizeof(index));
     //Сохраняем картинку
     fs.CopyFrom(ms, ms.Size);
     ms.Free;
    end;
  finally
   fs.Free;
  end;
end;
```

Все достаточно просто, если посмотреть на комментарии. Для работы с файлом используется объект TFileStream. Во время инициализации, когда мы открываем файл, в качестве параметра открытия указывается значение fmCreate, что заставляет систему всегда создавать файл заново. Даже если файл уже существовал, то он будет уничтожен и создан заново:

```
fs:=TFileStream.Create(FileName, fmCreate);
```

Первым делом в файл лучше всего записать какую-нибудь сигнатуру. При открытии программа прочитает эту сигнатуру, и если она отличается от записанной, то программа не будет пытаться открыть файл, поскольку он может иметь другой формат, иначе могут возникнуть ошибки открытия. В нашем примере в качестве сигнатуры я записываю четыре символа (proj):

```
fs.Write('proj', 4);
```

Теперь сохраняем количество компонентов, находящихся на форме:

```
index:=plClient.ComponentCount;
fs.Write(index, sizeof(index));
```

Переходим к сохранению всех компонентов. Для этого запускается цикл перебора всех компонентов на панели. Внутри цикла сохраняем левую и верхнюю позиции. Здесь все достаточно просто, потому что это числовые параметры.

А как же сохранить изображение? Тут все сложнее. Я не раз получал письма, в которых пользователи, описывая свои попытки сохранить изображение, приводят строку типа:

```
fs.Write(Image1.Picture.Bitmap,
      sizeof(Image1.Picture.Bitmap));
```

Все сохраняется и загружается успешно, а ошибки происходят уже после загрузки. Bitmap — это объект, а все переменные объектов в Delphi представляют собой указатели. Получается, что приведенный фрагмент кода сохраняет в файле указатель на объект в памяти. Во время загрузки мы снова получаем этот указатель, но по указанному адресу располагаются уже другие данные (если они вообще там есть). Поэтому когда программа пытается интерпретировать эти данные как изображение, естественно происходит ошибка.

Таким образом, код сохранения объекта, в том числе изображения, должен как минимум дополняться следующей строкой:

```
fs.Write(Image1.Picture.Bitmap^, Размер);
```

В первом параметре мы сохраняем то, что находится по адресу Image1.Picture.Bitmap. Для этого указан символ ^. Но вот определить размер сохраняемых данных сложно. Есть один способ определения размера, но мы его рассмотрим чуть позже. А на данном этапе я рекомендую использовать другой способ сохранения изображения, который показан в листинге 2.3. Для начала создаем поток данных в памяти и сохраняем в нем изображение. Не забывайте, что после сохранения текущая позиция в потоке устанавливается на конец памяти и необходимо сдвинуть позицию на начало, чтобы потом копировать данные из потока в памяти в файл с самого начала.

Однако сначала мы сохраняем размер изображения (его можно найти через свойство Size потока данных), чтобы потом при чтении можно было определить, сколько нужно прочитать данных изображения. А вот после этого мы копируем в файл данные изображения из памяти методом CopyFrom.

Еще один способ определения размера выглядит следующим образом.

1. Записываем в файл переменную типа Integer. Пока она может быть пустой, потому что это не имеет никакого значения. Запись нужна только для того, чтобы зарезервировать место в файле.

2. Запоминаем текущую позицию в файле — это будет начало изображения. Определить эту позицию можно путем вызова метода seek, в качестве смещения указать 0, чтобы не было смещения, а точкой отсчета сделать текущую позицию soFromCurrent. Таким образом, смещения не будет и функция вернет нам текущую позицию.
3. Сохраняем изображение.
4. Определяем текущую позицию в файле. Это значение минус первая запомненная позиция (начало записи изображения) как раз и представляет собой размер.
5. Переходим в позицию начала изображения. Это значение мы запомнили на втором шаге, и теперь не составляет труда перейти к нему методом seek. Кроме того, перемещаемся на размер числа типа Integer в сторону начала (seek(-sizeof(Integer), soFromCurrent)), чтобы оказаться в начале записанного числа заготовки для размера изображения, и на его место записываем определенный ранее размер.

Если не удается определить размер записываемых данных другими способами, поскольку объем данных заранее не известен, то можно использовать этот способ. Например, допустим, что нужно сохранить в файле приходящие по сети данные. Сколько их будет, мы не знаем, поэтому резервируем место, а потом записываем все, что приходит по сети. Потом после получения всех данных можно будет записать точное число.

Код, реализующий все вышесказанное, представлен в листинге 2.4.

Листинг 2.4. Пример определения размера записанных данных

```
var
 index, startpos, endpos : integer;
// Записываем в файл ноль, здесь будет потом размер данных
begin
 index:=0;
 fs.Write(index, sizeof(index));
 // Определяем текущее положение в файле
 startpos := fs.Seek(0, soFromCurrent);
 // Записываем картинку
 fs.Write(Bitmap^, Размер картинки);
 // Определяем текущее положение в файле
 endpos := fs.Seek(0, soFromCurrent);
 // Перемещаемся на начало записанного числа
 endpos := fs.Seek(startpos-sizeof(index), soFromBeginning);
 // Записываем размер данных
 index:=endpos-startpos;
 fs.Write(index, sizeof(index));
end;
```

Загрузка данных должна происходить в обратном порядке. Сначала проверяем наличие в файле сигнатуры. Если ее нет, это означает, что формат файла неправильный. Если все нормально, то читаем количество компонентов в файле, последовательно загружаем их и сразу же создаем компоненты TImage с размещением их на панели главной формы программы.

Листинг 2.5. Загрузка файла

```
procedure TSaveProjectForm.acOpenExecute(Sender: TObject);
var
 fs : TFileStream;
 Sign : array[0..3] of char;
 Count, i, index : Integer;
 imgTemp:TImage;
 ms: TMemoryStream;
begin
 // Запрашиваем имя файла
 if not OpenDialog1.Execute() then
  exit;
 FileName:=OpenDialog1.FileName;

 // Читаем проект
 fs:=TFileStream.Create(FileName, fmOpenRead);
 try
  // Записываем сигнатуру
  fs.Seek(0, soFromBeginning);
  fs.Read(Sign, 4);
  if sign<>'proj' then
   begin
    Application.MessageBox('Сигнатура не соответствует файлу проекта.',
    'Ошибка открытия', MB_OK+MB_ICONERROR);
    fs.Free;
    exit;
   end;

  fs.Read(Count, sizeof(Count));
  for i:=0 to Count-1 do
   begin
    imgTemp:=TImage.Create(plClient);

    // Левая позиция
    fs.Read(index, sizeof(index));
    imgTemp.Left:=index;

    // Верхняя позиция
    fs.Read(index, sizeof(index));
    imgTemp.Top := index;

    // Картинка
    fs.Read(index, sizeof(index));
    ms:=TMemoryStream.Create;
    ms.CopyFrom(fs, index);
    ms.Seek(0, soFromBeginning);
    imgTemp.Picture.Bitmap.LoadFromStream(ms);
    ms.Free;

    // Устанавливаем основные параметры компонента
    imgTemp.Autosize:=true;
    imgTemp.Transparent:=true;
    plClient.InsertControl(imgTemp);
   end;
 finally
  fs.Free;
 end;
end;
```

В листинге 2.5 реализована процедура загрузки файла. Я снабдил листинг подробными комментариями, чтобы вам проще было разобраться. Самое интересное находится в коде загрузки изображения. Сначала считываем размер, а потом читаем из файла данные в поток данных `TMemoryStream`. Прежде чем загрузить данные в объект `Bitmap`, необходимо переместиться в начало потока в памяти.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\SaveProject.

Общие сведения о формате XML

Когда появилась технология XML, я воспринял ее как очередной маркетинговый ход некоторых крупных компаний, потому что не увидел ее явных преимуществ перед HTML. Да, это открытый формат, который может использоваться для переноса данных между приложениями. Но ведь документ, сохраненный в MS Word, не обязательно правильно откроется в другой программе, поскольку каждый производитель может применять собственные теги.

Слишком «мягкие» стандарты я воспринимаю с опаской, а XML как раз и является мягким стандартом, потому что позволяет разработчику создавать документы, теги в которых никому кроме него не известны. Но однажды я писал программу, предназначенную для работы с Интернет-приложениями, и на начальном этапе было принято решение хранить данные в формате XML. В этот момент я ощутил все прелести этого формата, потому что хранение данных в XML позволило решить множество проблем.

Главная проблема, с которой сталкивается каждый разработчик, — формат хранения данных. Допустим, вы разработали программу, для хранения данных в которой использовалась структура `ComponentProp`, но впоследствии оказалось, что в нее необходимо добавить еще одно поле. Однако после добавления поля размер структуры увеличивается, поэтому при загрузке файла, созданного в предыдущей версии программы, возникнут ошибки. В исходном файле размер структуры может составлять 100 байт, а новая версия прочитает больше данных, и весь процесс загрузки нарушится.

Единственное решение — ввести в заголовок файла параметр, который будет хранить версию файла, и создавать загрузчики файлов для каждой версии. Поддержка такой программы будет сложной, и я не раз убеждался в том, что необходимо более эффективное решение.

Таким решением для меня стал формат XML. При его использовании у меня не было проблем с версиями, возникала только проблема с хранением бинарных данных, таких как изображения. Давайте рассмотрим пример, в котором программа сохраняет все необходимые данные в формате XML. Вы все увидите на практике.

Но для начала разберемся с форматом XML и узнаем, что он собой представляет, чтобы можно было написать собственную программу создания и загрузки XML-файлов. Если вы знакомы с языком разметки HTML, то с XML разобраться будет достаточно просто.

Файл в формате XML состоит из данных, которые заключаются в теги. Тег можно воспринимать как объединение имени и параметров данных, заключенное в угловые скобки. В самом начале файла идет тег, который описывает версию и кодировку файла. Это необходимо, чтобы программа анализатора смогла правильно представлять данные. Например:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

В этой строке тег заключается не только в угловые скобки, но и окружается знаками вопроса. Таким образом оформляются служебные теги. Имя этого тега `xml`, затем идут параметры, задающие номер версии и кодировку. Параметры в теге описываются в виде:

```
имя="значение"
```

После заголовка располагаются теги, описывающие сам документ. Каждый тег является парным, то есть состоит из открывающего и закрывающего тегов:

```
<имя> Значение </имя>
```

Имена обоих тегов одинаковы, но перед именем закрывающего тега стоит знак `/` (слеш). Между тегами располагается значение. Если вы работали с языком HTML, то уже не раз использовали такую конструкцию. Например, в HTML есть тег `<B>`, который делает текст жирным:

```
<b> Этот текст будет жирным </b>
```

Теги документа должны быть заключены в один большой блок. Например:

```
<DOCUMENT>
  <PROPERTIES1>Свойство 1</PROPERTIES1>
  <PROPERTIES2>Свойство 2</PROPERTIES2>
</DOCUMENT>
```

Здесь два тега объединены в один большой тег `<DOCUMENT>`, который называют корневым. В файле не может быть двух корневых тегов. Например, следующий фрагмент неправилен:

```
<DOCUMENT>
  <PROPERTIES1>Свойство 1</PROPERTIES1>
  <PROPERTIES2>Свойство 2</PROPERTIES2>
</DOCUMENT>
<DOCUMENT>
  <PROPERTIES3>Свойство 3</PROPERTIES3>
</DOCUMENT>
```

Здесь два корневых тега, а должен быть один. В HTML-файле весь документ также заключается в один корневой тег — `<html>`.

Все названия тегов вы можете задавать сами, и тут никаких жестких требований нет. Именно это долгое время меня отпугивало от использования формата XML, потому что каждый производитель может называть одни и те же вещи по-своему.

В файле можно указывать и комментарии, которые игнорируются анализатором файла. Комментарий выглядит следующим образом:

```
<!– Это комментарий –>
```

В начале строки располагаются символы `<!–`, а завершает комментарий последовательность символов `–>`. Такое оформление комментариев похоже на комментарии в языке разметки HTML.

Давайте рассмотрим все сказанное на практике и увидим, какие преимущества дает нам формат XML перед традиционным форматом хранения файлов. Для этого не будем придумывать новый пример, а изменим код сохранения и загрузки файлов в программе, которую мы рассматривали в разделе «Сохранение данных проекта».

Для реализации примера мы напишем собственный код создания и загрузки файлов. В Интернете сейчас можно найти достаточно много универсальных компонентов, облегчающих работу с XML, но их возможности нам не понадобятся. Чаще приходится решать более простые задачи, и для них лучше создавать собственный код. В примере изображение сохраняться не будет. Вместо этого я буду сохранять его индекс в списке TImageList. Попробуйте сами реализовать возможность сохранения бинарных файлов.

Итак, код сохранения проекта в формате XML представлен в листинге 2.6.

Листинг 2.6. Сохранение проекта в XML-файле

```
procedure TSaveProjectForm.acSaveExecute(Sender: TObject);
var
 i:Integer;
 tfFile: TextFile;
begin
 // Если имя файла не задано, то вызвать "Сохранить как"
 if FileName='' then
  begin
   acSaveAsExecute(nil);
   exit;
  end;

 if FileExists(FileName) then
  if FileGetAttr(FileName) and faReadOnly > 0 then
    begin
     Application.MessageBox('Нельзя сохранить проект',
         'Ошибка', MB_ICONINFORMATION+MB_OK);
     exit;
    end;

 // Открыть файл
 AssignFile(tfFile, FileName);
 // Доступ к файлу только для чтения
 FileMode := 1;
 // Очистить файл
 Rewrite(tfFile);
 // Проверка выполнения команд
 i:=IOResult;
 if i <> 0 then
  begin
   Application.MessageBox('Не получается сохранить проект',
     'Ошибка', MB_ICONINFORMATION+MB_OK);
   Exit;
  end;

 try
```

продолжение

Листинг 2.6 (продолжение)

```
    // Запись заголовка
    Writeln(tfFile, '<?xml version="1.0" encoding="iso-8859-1"?>');
    Writeln(tfFile, '');
    Writeln(tfFile, '<!- Создал Фленов Михаил(http://www.vr-online.ru) ->');
    Writeln(tfFile, '');
    Writeln(tfFile, '<COMPONENTS>');

    // Запись компонентов
    for i:=0 to plClient.ComponentCount-1 do
     begin
      Writeln(tfFile, '<COMPONENT>');

      // Левая позиция
      Writeln(tfFile, '<LEFT>' +
         IntToStr(TImage(plClient.Components[i]).Left)+
         '</LEFT>');

      // Верхняя позиция
      Writeln(tfFile, '<TOP>' +
         IntToStr(TImage(plClient.Components[i]).Top)+
         '</TOP>');

      // Верхняя позиция
      Writeln(tfFile, '<IMAGE_INDEX>' +
         IntToStr(TImage(plClient.Components[i]).Tag)+
         '</IMAGE_INDEX>');

      // Закрываем тег компонента
      Writeln(tfFile, '</COMPONENT>');
     end;

  finally
   // Закрываем корневой тег файла
   Writeln(tfFile, '</COMPONENTS>');
   CloseFile (tfFile);
  end;
 end;
```

Так как XML-файл является текстовым, то намного удобнее будет открывать его через объект `AssignFile`. Объект `TFileStream`, который мы использовали ранее, лучше подходит для бинарных данных. Открыв файл, мы переводим его в режим только для записи (`FileMode:= 1`) и очищаем его содержимое (`Rewrite(tfFile)`):

```
AssignFile(tfFile, FileName);
FileMode := 1;
Rewrite(tfFile);
```

Теперь записываем заголовок, в котором находится строка, описывающая версию и язык XML-файла. Далее записываем комментарий. Его можно использовать для хранения сигнатуры файла. Например, если третья строка в файле не содержит нужного комментария, то этот будет означать, что XML-файл создан в другой программе и при его открытии могут возникнуть проблемы.

Далее запускается цикл, в котором каждый элемент записывается в файл. Параметры каждого изображения, установленного на форме, заключаются в тег `<COMPONENT>`.

Внутри него идут теги, описывающие свойства изображения (расположение и индекс изображения в списке).

Для тестирования правильности сохранения файла не обязательно писать загрузчик, достаточно просто запустить XML-файл, чтобы он открылся в Internet Explorer (рис. 2.3). Если вы увидите свои данные, значит, файл создан верно. Если в окне будет сообщение об ошибке в XML-коде сохранения данных, то сначала нужно исправить эту ошибку, а потом уже продолжать работу, иначе во время загрузки может произойти что-нибудь фатальное для программы.

В Internet Explorer очень удобно анализировать содержимое XML-файла. Браузер выделяет элементы, и их можно сворачивать так же, как иерархические древовидные структуры.

Теперь переходим к рассмотрению процесса загрузки. У нас уже есть текстовый файл в формате XML, который необходимо загрузить. Некоторые программисты боятся анализа текстовых файлов. Я сам раньше боялся и не понимал, как программы в ОС Linux хранят всю конфигурацию в текстовых файлах. Но возможности, которые предоставляет текст, позволили преодолеть боязнь, и теперь я пишу для каждой программы собственный загрузчик и анализатор XML-файлов.

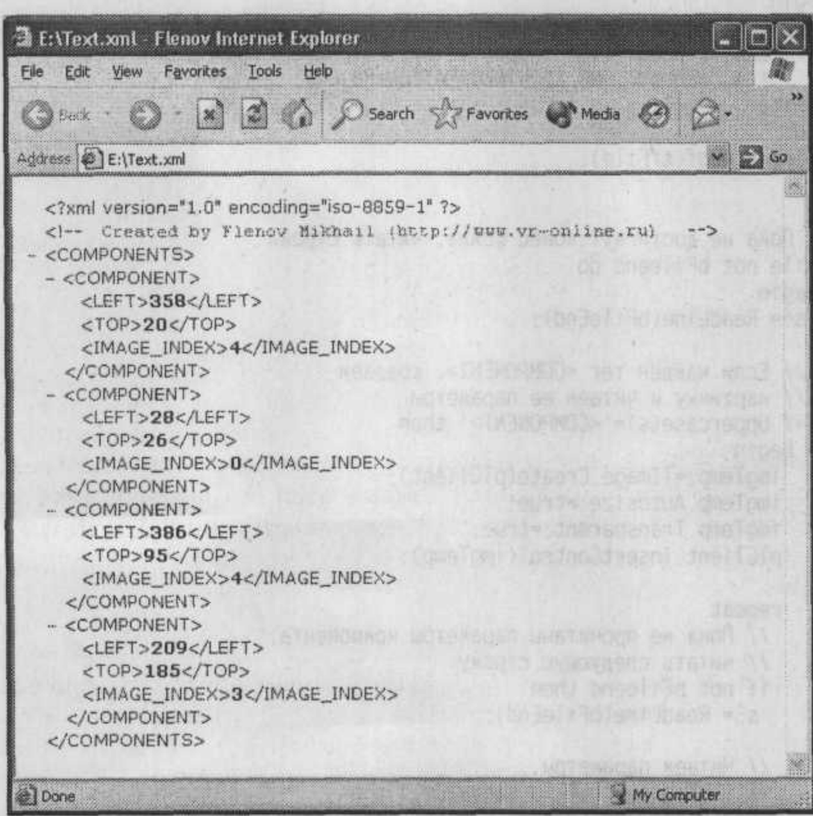


Рис. 2.3. Просмотр XML-файла в Internet Explorer

В листинге 2.7 показан пример анализатора языка XML, который восстанавливает сохраненное содержимое проекта.

Листинг 2.7. Загрузка XML-файла

```
procedure TSaveProjectForm.acOpenExecute(
  Sender: TObject):
var
 imgTemp : TImage;
 bFileend : boolean;
 s,tagvalue : String;
 bmpTemp : TBitmap;
begin
 // Запрашиваем имя файла
 if not OpenDialog1.Execute() then
  exit;
 FileName:=OpenDialog1.FileName;

 AssignFile(tfFile, OpenDialog1.FileName);
 FileMode := 0;
 Reset (tfFile);
 {$I+}
 if IOResult <> 0 then
  begin
   Application.MessageBox(PChar(
    'Can''t read file '+ OpenDialog1.FileName +
    '!'), 'Error', MB_ICONINFORMATION+MB_OK);
   exit;
  end;
 bFileend:=eof(tfFile);

 try
  // Пока не достигнут конец файла, читать строки
  while not bFileend do
   begin
    s:= ReadLine(bFileEnd);

    // Если найден тег <COMPONENT>, создаем
    // картинку и читаем ее параметры
    if Uppercase(s)='<COMPONENT>' then
     begin
      imgTemp:=TImage.Create(plClient);
      imgTemp.Autosize:=true;
      imgTemp.Transparent:=true;
      plClient.InsertControl(imgTemp);

      repeat
       // Пока не прочитаны параметры компонента,
       // читать следующую строку
       if not bFileend then
        s:= ReadLine(bFileEnd);

       // Читаем параметры
       if (IsTag(s, 'LEFT', tagvalue)) then
        begin imgTemp.Left:=StrToInt(tagvalue); Continue; end;
```

```
      if (IsTag(s, 'TOP', tagvalue)) then
       begin imgTemp.Top:=StrToInt(tagvalue); Continue; end;
      if (IsTag(s, 'IMAGE_INDEX', tagvalue)) then
       begin
        bmpTemp:=TBitmap.Create;
        ImageList2.GetBitmap(StrToInt(tagvalue), bmpTemp);
        imgTemp.Picture.Bitmap.Assign(bmpTemp);
        bmpTemp.Free;
        Continue;
        imgTemp.Tag:=StrToInt(tagvalue);
       end;
     until (bFileend) or (uppercase(s)='</COMPONENT>');
    end;
   end;
  finally
   CloseFile(tfFile);
  end;
 end;
```

Для чтения используем тот же метод доступа к файлу. Сразу после его открытия включаем режим доступа «только для чтения», чтобы случайно не натворить бед.

При загрузке данных нужно быть аккуратным и постоянно проверять достижение конца файла. Если во время записи происходит ошибка и запись прерывается на каком-либо параметре, то результат чтения может оказаться неверным. Чтобы иметь возможность загрузить хоть какие-то данные, нужно вставлять проверочный код везде, где только можно.

Для чтения строки из файла я написал небольшую функцию ReadLine (листинг 2.8). При ее написании я столкнулся с одной проблемой — тип переменной TextFile, который используется для хранения идентификатора файла, нельзя передавать в качестве параметра в процедуры и функции, а моей функции необходимо «знать» идентификатор. Для исправления ситуации переменная, предназначенная для хранения идентификатора, должна быть объявлена глобальной или в качестве свойства объекта формы. Я выбрал второй вариант, потому что глобальные переменные в объектно-ориентированном программировании использовать не рекомендуется.

Листинг 2.8. Функция чтения строки из файла

```
function TSaveProjectForm.ReadLine(
  var MIEof: boolean): string;
var
  s : string;
begin
  Readln(tfFile,s);
  MIEof:=eof(tfFile) and (FilePos(tfFile) = FileSize(tfFile));
  Result:=Trim(s);
end;
```

Прочитав строку, мы должны проверить, есть ли в ней тег. Так как таких проверок должно быть очень много, я написал функцию IsTag (листинг 2.9). Функции нужно передать строку и имя тега, который мы хотим найти в строке. Если тег

есть и имеет формат *<имя>значение</имя>*, то функция возвращает через третий параметр значение, которое записано в теге.

Листинг 2.9. Проверка на присутствие тега

```
function TSaveProjectForm.IsTag(sLine, sTagName: string;
  var sTagValue: string): boolean;
begin
 if AnsiUpperCase(Copy(sLine,1,Length(sTagName)+2)) = '<'+sTagName+'>' then
  begin
   Delete(sLine,1,Length(sTagName)+2);

   if Copy(sLine,Length(sLine) - (Length(sTagName)+2),Length(sTagName)+3) =
         '</'+sTagName+'>' then
    Delete(sLine, Length(sLine) - (Length(sTagName)+2), Length(sTagName)+3);

   sTagValue:=sLine;
   Result:=True
  end
 else
  Result:=False;
end;
```

Таким образом, для чтения нашего файла проекта необходимо последовательно читать строки, а если найдена строка с тегом <COMPONENT>, это означает, что далее идет описание изображения. Для получения параметров изображения запускается цикл, который читает файл дальше, пока не обнаруживает строку с закрывающим тегом </COMPONENT>. В этом цикле читаются все найденные параметры, и на их основе создается изображение (компонент TImage).

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\XML.

При использовании формата XML очень удобно расширять возможности программы. Например, если вы решите сохранять в файле новое поле, определяющее ширину изображения, просто заведите для этого отдельный тег. Добавив код сохранения и загрузки тега, не нужно заботиться о версиях файла. При загрузке старых проектов тег ширины изображения найден не будет, поэтому будет использоваться значение по умолчанию. Таким образом, один и тот же код сможет загружать и новые, и старые файлы с новым тегом и без него.

В результате мы получаем отличную масштабируемость и совместимость не только вниз (со старыми версиями), но и вверх (с новыми версиями). Допустим, старая версия программы, которая «не знает» о существовании тега ширины, пытается загрузить файл, в котором есть этот тег. Во время загрузки никакой ошибки не произойдет, потому что тег будет проигнорирован и загрузятся только те данные, о которых «знает» программа.

Чтобы не нарушить совместимость, старайтесь не переименовывать теги в процессе расширения возможностей программ. Лучше завести новое имя тега, а старое просто не использовать.

XML в Delphi

Хотя самостоятельно создать и прочитать XML-файл достаточно просто, этот формат оказывается эффективным только для решения простых задач. На самом деле формат XML намного сложнее, и для обработки документа приходится писать весьма сложные и универсальные процедуры, поэтому я пользуюсь им только для хранения и загрузки данных своих проектов.

Но не все так страшно, потому что компания Borland всегда старается облегчить жизнь разработчикам. В Delphi есть очень удобный компонент (класс) `TXMLDocument`, который упрощает работу с XML-документами. Этот компонент поддерживается как в версии для Win32, так и для .NET. Загрузив документ в класс, вы можете работать с ним так же, как с деревом (компонентом `TTreeView`).

По правде говоря, компонент `TXMLDocument` документирован в файле помощи очень плохо. Нет, там описаны все поля и методы, но когда я стал разбираться с работой этого компонента, то столкнулся с множеством проблем, а по описанию из файла помощи очень сложно что-либо понять о назначении свойств.

Главная проблема при работе с компонентом `TXMLDocument` состоит в том, что во время отладки не видно значений свойств. Если прервать выполнение программы и попытаться просмотреть значения тех или иных свойств, то ничего, кроме сообщения об ошибке доступа, не появится. Поэтому при создании программ для просмотра значений свойств приходится писать дополнительный код, призванный отображать нужные значения в окнах диалога.

Анализ документа с помощью компонента TXMLDocument

Давайте создадим приложение, которое сможет работать с любыми XML-документами, используя компонент `TXMLDocument`. А в процессе разбора приложения поближе познакомимся с форматом XML. Для этого в главном окне программы примера (рис. 2.4) нам понадобятся:

- меню и панель с кнопками **Создать**, **Открыть**, **Сохранить**;
- дерево (компонент `TTreeView`), в котором можно отображать структуру XML-документа;

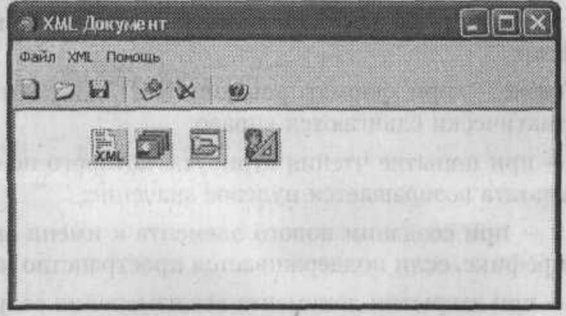


Рис. 2.4. Главное окно будущей программы

- список изображений (компонент TImageList), чтобы окно программы выглядело более эстетичным;
- ну и конечно же, компонент TXMLDocument, с помощью которого будет происходить анализ документа.

Но прежде чем приступать к программированию, давайте посмотрим, какие свойства есть у компонента TXMLDocument.

- Active — когда мы делаем компонент активным, то он анализирует документ и разбивает его по тегам.
- DOMVendor — программа, которая будет заниматься разбором XML-документа. Сам компонент этим не занимается, а использует промежуточную утилиту. По умолчанию выбрана программа MSXML, но можно выбрать Xerces XML или Open XML. Все утилиты бесплатные, но я рекомендую использовать MSXML. Дело в том, что эта утилита должна устанавливаться на машине клиента, на которую вы будете устанавливать свою программу, а MSXML имеется на большинстве компьютеров с Windows, а ведь именно для этой платформы мы пишем программу.
- Filename — имя файла, которое связано с XML-документом.
- NodeIntendStr — количество пробелов, которые вставляются для удобочитаемости XML-документа. Формат XML имеет древовидную структуру, и отступы, создаваемые с помощью пробелов, позволяют сделать код более понятным.

  Посмотрите на следующий пример:

  ```
  <COMPONENTS>
    <COMPONENT>
      <LEFT>358</LEFT>
      <TOP>20</TOP>
      <IMAGE_INDEX>4</IMAGE_INDEX>
    </COMPONENT>
  </COMPONENTS>
  ```

  Этот код намного удобнее воспринимать, потому что каждый уровень иерархии имеет свой отступ от левого края. Если бы отступов не было, то восприятие усложнилось бы. Это особенно заметно, когда размер документа исчисляется сотнями килобайт.
- Options — параметры XML-документа:
  - doNodeAutoCreate — если приложение пытается обратиться к элементу по имени, которого не существует, то элемент с указанным именем создается автоматически;
  - doNodeAutoIndent — при форматировании XML-документа дочерние элементы автоматически сдвигаются вправо;
  - doAttrNull — при попытке чтения атрибута, которого не существует, в качестве результата возвращается нулевое значение;
  - doAutoPrefix — при создании нового элемента к имени автоматически добавляется префикс, если поддерживается пространство имен URI;
  - doAutoSave — при закрытии документа все изменения сохраняются автоматически.

- ParseOptions — параметры анализатора XML-документа.
- XML — свойство типа TStrings, в котором находится XML-документ.

Теперь перейдем к созданию самого кода примера. Начнем с загрузки XML-документа. По щелчку на кнопке Открыть будет выполняться код из листинга 2.10.

Листинг 2.10. Открытие документа

```
procedure TXMLDocExampForm.acOpenExecute(Sender: TObject);
var
 root: IXMLNode:
 TempNode:TTreeNode;
begin
 if not OpenDialog1.Execute then
  exit;
 XMLDocument1.LoadFromFile(OpenDialog1.FileName);

 // Определение корневого элемента
 Root:=XMLDocument1.DocumentElement;

 // Добавление корневого элемента в дерево TTreeView
 TempNode:=TreeView1.Items.Add(nil, 'XML документ');
 TempNode.ImageIndex:=6;
 TempNode.SelectedIndex:=7;
 TempNode.Data:=TXMLNode(Root);

 // Загрузка уровня
 LoadLavel(TempNode, Root.ChildNodes);
end;
```

Чтобы удобней было разбирать пример, в листинге 2.11 я воспроизвел код XML-документа, описанного в разделе «Общие сведения о формате XML». Будем анализировать пример на основе этого документа.

Листинг 2.11. Пример XML-документа

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!- Created by Flenov Mikhail (http://www.vr-online.ru) ->

<COMPONENTS>
  <COMPONENT>
    <LEFT>358</LEFT>
    <TOP>20</TOP>
    <IMAGE_INDEX>4</IMAGE_INDEX>
  </COMPONENT>
  <COMPONENT>
    <LEFT>28</LEFT>
    <TOP>26</TOP>
    <IMAGE_INDEX>0</IMAGE_INDEX>
  </COMPONENT>
  <COMPONENT>
    <LEFT>386</LEFT>
    <TOP>95</TOP>
    <IMAGE_INDEX>4</IMAGE_INDEX>
  </COMPONENT>
</COMPONENTS>
```

Сначала появляется окно открытия файла. Если файл не выбран, то выходим из процедуры, иначе нужно загрузить XML-документ.

Для загрузки документа используем метод `LoadFromFile` компонента `TXMLDocument`. Этот метод загружает файл и сразу устанавливает свойство `Active` в `true`, чтобы анализатор проверил и разобрал файл по тегам. Если для загрузки документа использовать метод `LoadFromFile` свойства `XML`, то активировать компонент придется самостоятельно:

```
XMLDocument1.XML.LoadFromFile(OpenDialog1.FileName);
XMLDocument1.Active:=true;
```

Теперь можно было бы начать запрашивать у компонента теги, находящиеся в свойстве `XMLDocument1.ChildNodes`. Но это не очень хорошее решение, потому что данное свойство вернет нам все теги, включая информационные, а также комментарии. Теги лучше всего получать, начиная с корневого. Например, в листинге 2.11 корневым является тег `COMPONENTS`, а все остальные теги являются его подчиненными. И именно этот тег мы должны выявить, чтобы начать разбор с него.

Для выявления корневого тега используется свойство `DocumentElement`. Это свойство указывает на интерфейс типа `IXMLNode`. Все элементы XML-документа в компоненте `TXMLDocument` имеют этот тип. Обратите внимание, что первой в имени типа стоит буква I, а не привычная для объектов в Delphi буква `T`. Дело в том, что это интерфейс. В Delphi есть класс `TXMLNode`, но в большинстве случаев методы и свойства компонента `TXMLDocument` используют именно интерфейс, а не объект.

Интерфейсы некоторых программистов пугают, да и сам я не очень люблю их использовать, но применение интерфейса — единственный способ наделить объект возможностью наследования от нескольких объектов одновременно. Если вы не знакомы с интерфейсами, то можете не забивать себе голову лишними терминами, а просто помните, что работа с ними не слишком отличается от работы с обычными объектами.

После получения корневого элемента XML-документа создадим в компоненте `TTreeView` элемент, в который будут записываться все теги. Его можно было бы назвать так же, как и корневой элемент, но я выбрал название `'XML-документ'`:

```
TempNode:=TreeView1.Items.Add(nil, 'XML-документ');
TempNode.ImageIndex:=6;
TempNode.SelectedIndex:=7;
```

Для придания программе элегантности элементу присваиваются изображения папок. В моем списке `TImageList` эти изображения находятся под индексами 6 и 7.

Установив изображения, в свойстве `Data` нового элемента дерева сохраняем указатель на корневой элемент:

```
TempNode.Data:=TXMLNode(Root);
```

Обратите внимание, что переменная `root` имеет тип `IXMLNode`, то есть является интерфейсом. Свойство `Data` элемента дерева имеет тип указателя и может использоваться для хранения собственных данных. В указателе нельзя сохранить интерфейс, поэтому приходится преобразовывать его в объект путем явного указания типа:

```
TXMLNode(root)
```

Зачем мы сохраняем указатель в элементе дерева? Об этом вы узнаете чуть позже. Сейчас мы закладываем фундамент, который потребуется в ближайшем будущем, когда мы будем писать код редактирования элементов.

После этого вызывается процедура LoadLavel, реализацию которой нужно добавить в наш проект (листинг 2.12). В качестве параметров в процедуру передаются:

- элемент типа TTreeNode, к которому нужно добавлять найденные в XML-документе теги;
- элемент типа IXMLNode, у которого нужно найти дочерние элементы и добавить в дерево.

Листинг 2.12. Загрузка уровня

```
procedure TXMLDocExampForm.LoadLavel(
  TreeNode:TTreeNode; XMLNode:IXMLNodeList);
var
  i:Integer;
  TempNode:TTreeNode;
begin
  for i:=0 to XMLNode.Count-1 do
  begin
    if not XMLNode[i].HasChildNodes then
     begin
      // У текущего элемента нет дочерних, поэтому добавляем его как значение
      with TreeView1.Items.AddChild(TreeNode, XMLNode[i].NodeValue) do
       begin
        ImageIndex:=8;
        SelectedIndex:=8;
        TempNode.Data:=TXMLNode(XMLNode[i]);
       end;
     end
    else
     begin
      // У элемента есть дочерние, поэтому добавляем его как тег
      TempNode:=TreeView1.Items.AddChild(TreeNode, XMLNode[i].NodeName);
      TempNode.ImageIndex:=6;
      TempNode.SelectedIndex:=7;
      TempNode.Data:=TXMLNode(XMLNode[i]);

      // Загружаем дочерние элементы текущего тега
      LoadLavel(TempNode, XMLNode[i].ChildNodes);
     end;
  end;
end;
```

Давайте подробно рассмотрим процедуру LoadLavel, потому что в ней много «подводных камней». Процедура загружает элементы указанного уровня. Мы уже не раз говорили, что XML-документ имеет древовидную структуру, поэтому удобно иметь универсальный метод, который мог бы загрузить любой уровень.

Все тело процедуры загрузки представляет собой цикл, который перебирает все дочерние элементы текущего XML-элемента. Количество элементов находится в свойстве Count. Внутри цикла проверяем это свойство, и если у текущего элемента

нет дочерних элементов (свойство HasChildNodes равно false), то это означает, что мы уже добрались до значения и добавляем его в дерево TTreeView. Только у значения тега не может быть дочерних элементов. Значение находится в свойстве NodeValue.

Если найден тег, то есть свойство HasChildNodes равно true, то добавляем элемент как тег. Имя тега можно получить через свойство NodeName. После этого запускаем процедуру LoadLavel, чтобы найти и отобразить все дочерние элементы следующего уровня.

Здесь нужно сделать одно замечание по поводу свойств NodeValue и NodeName. Для значений свойство NodeName всегда является пустым. Для тегов свойство NodeValue является недоступным. При попытке обратиться к нему происходит ошибка, и процедура дальше уже не выполняется. Поэтому вы должны четко определять, какой элемент вам сейчас доступен, — тег или значение.

Чтобы лучше понять вышесказанное, запустите программу и загрузите какой-нибудь XML-документ, например созданный с помощью программы из листинга 2.11. Посмотрите на результат работы (рис. 2.5). Обратите внимание на значения в дереве. Они являются как бы подчиненными для своего тега. Именно так их воспринимает анализатор. Это немного усложняет работу программы в реальных условиях, но зато дает нам универсальное решение.

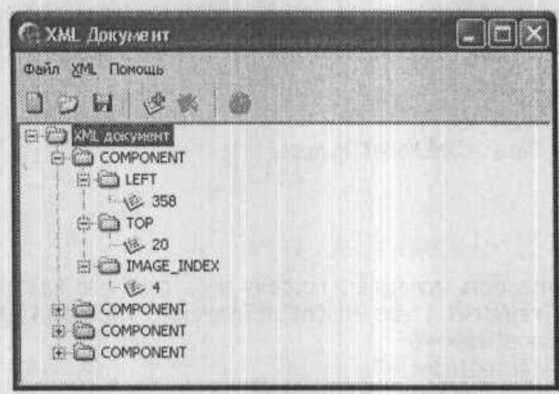


Рис. 2.5. Результат работы программы

В коде загрузки дерева XML-тегов очень интересной является следующая строка:

```
TempNode.Data:=TXMLNode(XMLNode[i]);
```

При добавлении в дерево нового элемента в свойстве Data мы сохраняем указатель на элемент в XML-документе. Это позволит в дальнейшем легко связать выделенный элемент в дереве TTreeView с элементом типа TXMLNode в компоненте TXMLDocument.

Обратите внимание, что переменная XMLNode имеет тип IXMLNode, то есть является интерфейсом. В данном случае снова приходится преобразовывать его в объект путем явного указания типа:

```
TXMLNode(XMLNode[i])
```

Свойство Data будем использовать при добавлении новых тегов, и тогда вы сможете по достоинству оценить возможности этого свойства.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\XMLDocument.

Атрибуты тега

Теперь выясним, что такое атрибуты тега. До сих пор я игнорировал эту тему, чтобы не особо забивать вам голову, ведь мы рассматривали простейшие XML-документы без каких-либо атрибутов.

Что такое атрибут? Каждый тег может иметь свойства, которые описываются в атрибутах. Рассмотрим следующий пример XML-документа:

```
<COMPONENTS>
    <COMPONENT Left=10 Top=20>
        <IMAGE_INDEX>4</IMAGE_INDEX>
    <COMPONENT>
<COMPONENTS>
```

В этом примере тег COMPONENT имеет два атрибута: Left и Top. Их объявление происходит прямо в фигурных скобках после имени тега.

Для доступа к атрибутам с помощью компонента TXMLDocument у каждого элемента TXMLNode есть свойство AttributeNodes. Сам доступ к атрибутам можно реализовать через их индексы или имена. Например, чтобы узнать значение атрибута Left, необходимо выполнить строку:

```
XMLNode.Attributes['Left'];
```

Изменение XML-документа

Какой смысл загружать XML-документ, если его нельзя редактировать? Давайте введем в программу возможность добавления новых тегов в произвольное место. Для этого вставим в окно кнопку **Добавить элемент**. По щелчку на этой кнопке будет выполняться код из листинга 2.13.

В самом начале проверяем, чтобы хоть какой-то элемент в дереве TreeView был выделен. Если ничего не выделено, то мы не сможем определить элемент, к которому нужно добавлять новый тег.

Следующим этапом проверяем, чтобы выделенный тег не был значением. К значению не стоит добавлять подчиненные теги, иначе могут возникнуть проблемы при анализе. Как определить, значение это или тег? В нашем примере это делается достаточно просто — по индексу используемого изображения. Для тегов у нас устанавливается индекс изображения 6, а для значений — 8.

Теперь запрашиваем имя и значение тега с помощью окна ввода параметра функции InputQuery. В реальном приложении необходимо будет создать собственную форму для ввода обоих значений в одном окне. В данном примере я не стал этого делать, чтобы не загромождать его лишними формами.

Листинг 2.13. Добавление дочернего тега к выделенному

```
procedure TXMLDocExampForm.acAddExecute(Sender: TObject);
var
 sTag, sTagValue:String;
begin
 // Если в дереве ничего не выделено, то добавлять некуда
 if TreeView1.Selected=nil then
  begin
   ShowMessage('Выделите элемент, к которому нужно добавить тег');
   exit;
  end;

 // Если выделен корневой элемент или значение, то добавлять нельзя
 if (TreeView1.Selected.ImageIndex=8) then
  begin
   ShowMessage('Нельзя добавлять теги к значению или к корневому');
   exit;
  end;

 // Запрашиваем имя тега
 if not InputQuery('Имя тега', 'Введите имя тега', sTag) then
  exit;
 if not InputQuery('Значение', 'Введите значение', sTagValue) then
  exit;

 // Добавление тега и значения
 with IXMLNode(TreeView1.Selected.Data).AddChild(sTag, -1) do
  begin
   NodeValue:=sTagValue;
  end;

 // Обновляем список
 UpdateXMLView;
end;
```

Теперь добавляем новый тег. Для этого сначала определяем выделенный узел `XMLNode`. Помните, мы сохраняли на него указатель в свойстве `Data`? Вот как раз здесь это и пригодится. Указатель на текущий элемент `XMLNode` можно узнать с помощью свойства `Data`, то есть следующим образом:

```
IXMLNode(TreeView1.Selected.Data)
```

Для добавления нового тега к `XMLNode` есть метод `AddChild`, которому необходимо передать два параметра:

- имя нового тега;
- положение, куда его необходимо вставить (можно явно передать индекс, например 0, чтобы новый тег стал первым в списке дочерних, а можно указать −1, чтобы новый элемент добавился в конец).

Метод `AddChild` возвращает указатель на новый элемент. Чтобы получить доступ к этому элементу и изменить его значение, я использовал конструкцию `with`:

```
with IXMLNode(TreeView1.Selected.Data).AddChild(sTag, -1) do
 begin
  NodeValue:=sTagValue;
 end;
```

Эта конструкция соответствует фразе: «Вместе с новым созданным элементом выполнять действия ...». У нас выполняется одно действие — изменяется свойство NodeValue, то есть значение тега.

Изменять значение в нашей программе необходимо. Дело в том, что при загрузке значения мы не проверяем его существование, а просто обращаемся к свойству NodeValue. Если значения у тега нет, то при обращении произойдет ошибка. Вы скажите, что необходимо добавить проверку, но я не делаю этого, чтобы в XML- файле не было пустых тегов. Каждый тег должен иметь значение, иначе его следует удалить.

После добавления нового элемента вызывается процедура UpdateXMLView (листинг 2.14). Эту процедуру я создал для того, чтобы можно было обновлять содержимое дерева TTreeView, заново перечитывая XML-документ. Иногда это действительно оказывается полезным.

Листинг 2.14. Обновление загруженного документа

```
procedure TXMLDocExampForm.UpdateXMLView;
var
 TempNode:TTreeNode;
begin
 TreeView1.Items.BeginUpdate;
 TreeView1.Items.Clear;
 TempNode:=TreeView1.Items.Add(nil, 'XML документ');
 TempNode.ImageIndex:=6;
 TempNode.SelectedIndex:=7;

 LoadLavel(TempNode, XMLDocument1.DocumentElement.ChildNodes);
 TreeView1.Items.EndUpdate;
end;
```

Если вы заботитесь о быстродействии и уверены, что пользователь не внесет в документ кардинальных изменений, то обновлять весь документ не стоит. Лучше изменить код добавления элемента, как показано в листинге 2.15.

Листинг 2.15. Добавление XML-элемента без перезагрузки данных

```
procedure TXMLDocExampForm.acAddExecute(Sender: TObject);
var
 sTag, sTagValue:String;
 newNode:IXMLNode;
 TempNode:TTreeNode;
begin
 ...
 Предварительные проверки
 ...

 // Добавление XML тега и значения
 newNode:=IXMLNode(TreeView1.Selected.Data).AddChild(sTag, -1);
 newNode.NodeValue:=sTagValue;

 // Добавление в дерево элемента для тега
 TempNode:=TreeView1.Items.AddChild(TreeView1.Selected, sTag);
```

продолжение ⇒

Сейчас у нас уже достаточно информации, чтобы можно было создать приложение второго типа. Попробуйте реализовать этот вариант самостоятельно. Если не получиться, то можете обратиться к компакт-диску и посмотреть мой вариант.

ПРИМЕЧАНИЕ

Исходный код примера находится на компакт-диске в каталоге Sources\ch02\XMLWithImage.

Список открывавшихся файлов

Сохранять надо не только данные, с которыми работает программа, но список последних открывавшихся файлов, с которыми работал пользователь. Это не просто дань моде, это действительно удобная возможность. Например, когда я работаю с Delphi, то неделями открываю один и тот же проект. Постоянно искать его на диске — достаточно утомительное занятие, намного проще найти его в подменю Reopen меню File среди последних открывавшихся проектов.

Если в вашем меню Файл не так уж много пунктов, то намного удобнее поместить ссылки на последние открывавшиеся файлы прямо в него, не создавая дополнительного подменю Reopen, как это сделано в программах из пакета MS Office. Именно такое меню мы сейчас и попробуем создать.

Вы наверно уже заметили, что я люблю компоненты Action, потому что они действительно удобны, а также позволяют создавать меню в стиле XP. Именно поэтому мы будем разрабатывать пример на основе этой технологии. Однако тут есть много подводных камней по сравнению с классическим меню (компонентом TMainMenu), где пункты меню достаточно легко создавать динамически.

Итак, создадим программу, которая будет работать с текстовыми файлами, загружать их, сохранять и хранить список файлов, с которыми работал пользователь.

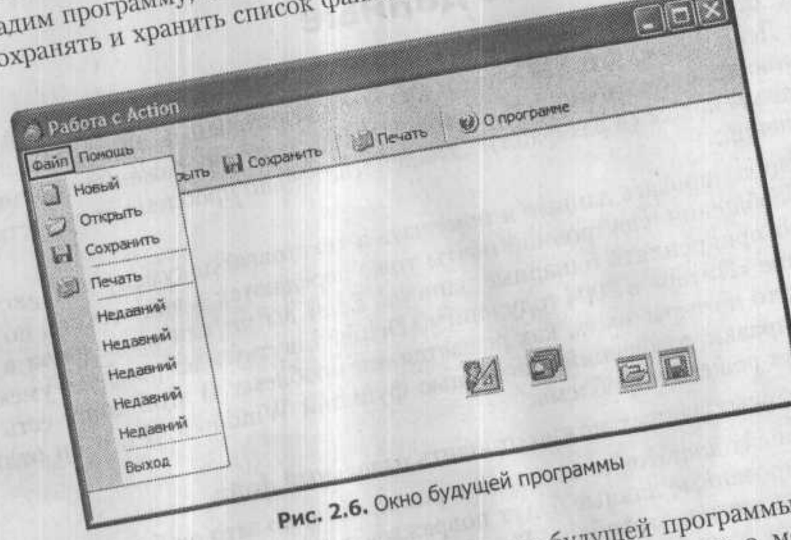


Рис. 2.6. Окно будущей программы

...исунке 2.6 можно видеть главную форму будущей программы ... изображение дает вам представление о м...

я специально показал открытым. Уже из рисунка можно догадаться, в чем будет секрет программы.

Итак, в диспетчере действий помимо необходимых команд **Новый**, **Открыть**, **Сохранить**, **Печать** и т. д. создадим пять пунктов с заголовком **Недавний** и именами от `acReOpenAction1` до `acReOpenAction5`. Одинаковое имя и последовательный номер в конце имени значительно упростит код программы.

По умолчанию все пять пунктов, которые будут ссылаться на последние открытые файлы, названы **Недавний**. Во время старта программы мы будем прятать пункты меню с таким заголовком. Однако заголовки всех пунктов будут храниться в файле, и прежде чем что-то прятать, эти заголовки нужно загрузить. Итак, для главной формы пишем реализующий все вышесказанное код, который будет выполняться по событию `OnCreate` (листинг 2.17).

Листинг 2.17. Обработчик события OnCreate

```
procedure TForm1.FormCreate(Sender: TObject);
var
 i: Integer;
 slFile:TStringList;
begin
 // Если файл существует, то загружаем пункты меню
 if FileExists(ExtractFilePath(Application.ExeName)+'menu.dat') then
  begin
   // Создать объект типа TStringList и загрузить в него имена
   slFile:=TStringList.Create;
   slFile.LoadFromFile(ExtractFilePath(Application.ExeName)+'menu.dat');

   // Скопировать все имена в 5 элементов типа TAction
   for i:=1 to 5 do
    TAction(FindComponent('acReOpenAction' +
                IntToStr(i))).Caption:=slFile[i-1];

   // Закрываем файл
   slFile.Free;
  end;

 // Обновить видимость пунктов меню
 UpdateReopenMenuVisability;
end;
```

При работе с файлами, прежде чем что-то открывать, я всегда проверяю существование файла. Любые попытки обращения к несуществующему файлу приведут к исключительным ситуациям или даже к краху программы.

Функции `FileExists` (как и методу `LoadFromFile`, который используется для загрузки файла) передается полный путь к файлу. Для его определения сначала узнаем путь к запущенной программе, то есть путь к папке, где находится исполняемый файл:

```
ExtractFilePath(Application.ExeName)
```

К этому пути добавляется само имя файла menu.dat.

Имена пунктов меню в файле хранятся в виде строк, и для загрузки таких файлов я предпочитаю использовать объект `TStringList`, с которым очень удобно работать. Для загрузки используется метод `LoadFromFile`.

Далее в цикле копируем загруженные пункты меню в элементы TAction. Самым последним вызывается метод UpdateReopenMenuVisability. В нем запускается цикл проверки всех пяти пунктов меню для загрузки последних файлов. Если имя элемента равно значению Недавний, то этот пункт скрывается (свойство Visible изменяется на false). Код этого метода представлен в листинге 2.18.

Листинг 2.18. Метод определения видимых пунктов меню

```
procedure TForm1.UpdateReopenMenuVisability;
var
  i: Integer;
begin
 for i := 1 to 5 do
  begin
   if TAction(FindComponent('acReOpenAction' + IntToStr(i))).Caption =
         'Недавний' then
    TAction(FindComponent('acReOpenAction' + IntToStr(i))).Visible := false
   else
    TAction(FindComponent('acReOpenAction' + IntToStr(i))).Visible := true;
  end;
end;
```

Когда нужно добавлять имена открытых в программе файлов в список? Однозначно при сохранении проекта. Если пользователь создал файл, то еще рано добавлять его имя в список, потому что имени пока нет. А вот после сохранения самое время. А что если пользователь открыл файл, просмотрел его, но не сохранил? Значит, имя такого файла надо добавить в список открывавшихся файлов сразу после открытия. Для этого нужно вызвать метод AddToReopen, которому будет передаваться имя открытого файла.

Реализацию метода AddToReopen можно увидеть в листинге 2.19.

Листинг 2.19. Реализация метода AddToReopen

```
procedure TForm1.AddToReopen(sName: String);
var
 i:Integer;
begin
 // Проверяем наличие файла в списке. Если есть, то выходим
 for i:=1 to 5 do
  if TAction(FindComponent('acReOpenAction'+IntToStr(i))).Caption=sName then
   exit;

 // Пересортировка пунктов меню
 for i:=5 downto 2 do
  begin
   if TAction(FindComponent('acReOpenAction'+IntToStr(i))).Caption=
        'Недавний' then
    TAction(FindComponent('acReOpenAction'+IntToStr(i))).Caption:=
     TAction(FindComponent('acReOpenAction'+IntToStr(i-1))).Caption;
  end;

 // Первый элемент из списка получает имя указанного файла
 acReOpenAction1.Caption:=sName;

 // Проверяем видимость элементов
```

```
 UpdateReopenMenuVisability;
end;
```

Для начала запускаем цикл перебора всех пяти элементов TAction, которые созданы для списка. В этом цикле выполняется проверка, находится файл в списке или нет. Если находится, то происходит выход из цикла (иначе пользователь может пять раз сохранить один и тот же файл, и имя этого файла заполнит весь список).

Следующим шагом пересортировываем список, сдвигая имена вниз, то есть первый элемент становится вторым, второй — третьим и т. д. Для этого удобен цикл с перебором от 5 до 2.

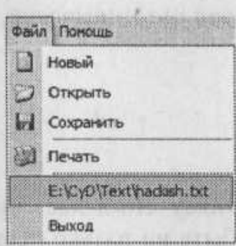


Рис. 2.7. Пример меню с одним пунктом в списке файлов

Первым элементом делаем имя указанного файла, то есть заменяем заголовок элемента полным путем к файлу (рис. 2.7). После этого вызываем метод UpdateReopenMenuVisability, который снова проверяет видимость элементов в списке (это необходимо, поскольку имена изменились).

По закрытии главной формы необходимо сохранить имена из списка последних открывавшихся файлов. Для этого служит код из листинга 2.20, который должен выполняться по событию OnClose. Для хранения имен я снова использую объект TStringList. В этом листинге, я думаю, все понятно и без дополнительных комментариев.

Листинг 2.20. Сохранение имен по событию OnClose

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
 i: Integer;
 slFile:TStringList;
begin
 // Создание объекта списка строк
 slFile:=TStringList.Create;
 // Запускаем цикл, в котором имена копируются в список
 for i:=1 to 5 do
  slFile.Add(TAction(FindComponent('acReOpenAction' +
   IntToStr(i))).Caption);

 // Сохранение файла
 slFile.SaveToFile(ExtractFilePath(Application.ExeName)+'menu.dat');
 // Освобождение переменной, то есть закрытие файла
 slFile.Free;
end;
```

Список последних открывавшихся файлов готов, но нужно добавить код, который позволит открывать файлы, представленные в этом списке. Для этого у всех пяти элементов TAction создаем один обработчик события. В нем нужно загрузить файл с именем, указанным в качестве заголовка выбранного пункта меню. На первый взгляд, определить имя несложно:

```
TAction(Sender).Caption
```

Однако не все так просто. Компоненты Action действуют по всем правилам Windows-программ, а в Windows-программах у каждого пункта меню в имени должна быть буква, обозначающая «горячую» клавишу, по нажатию которой можно выбрать этот пункт. Перед такой буквой устанавливается символ &. Delphi устанавливает этот символ автоматически, а значит, при чтении заголовка для имени файла:

```
c:\text\filename.txt
```

мы можем увидеть такой код:

```
&c:\text\filename.txt
```

Первый символ все портит, потому что делает путь нереальным. Самое неприятное, что этот символ может быть не только первым, но и находиться в любом месте пути. Чтобы его удалить, я использую функцию StringReplace, которой передаются четыре параметра:

- исходная строка, в которой необходимо удалить какой-либо набор символов (указываем TAction(Sender).Caption);
- набор удаляемых символов (строка '&');
- строка замены (поскольку нам нужно удалять, а не заменять, указываем пустую строку '');
- параметры замены.

По умолчанию при поиске учитывается регистр символов и заменяется только первый найденный символ. В нашем случае искомый символ не зависит от регистра и встречается только один раз. Но на всякий случай я указываю флаг [rfReplaceAll], благодаря которому можно удалить все вхождения символа &, если такое вдруг случится. Результатом будет строка, которая не содержит символов &.

Надеюсь, я все рассказал и ничего не забыл. Если что-то упустил, то всегда можно обратиться к исходному коду примера, находящемуся на компакт-диске. Так как он работает с файлом из каталога, из которого запускается программа, я рекомендую перед запуском скопировать код примера на жесткий диск, иначе будут проблемы с сохранением. Я не делал проверки на доступность файла на запись, а значит, при выходе из программы появятся ошибки.

Описанный код предлагает самое простое и элегантное решение проблемы. В поставке с Delphi есть пример, в котором делается то же самое, но его код намного сложнее. Элементы TAction создаются не в компоненте TActionManager, а в компоненте TActionList. В момент загрузки программы элементы TAction также изменяют заголовки, и если необходимо, то делаются видимыми. Подменю Reopen меню File, которое должно быть в вашей программе, ищется программно.

Зачем так усложнять задачу? У компонента TActionManager есть свойство FileName. Если в нем указать имя файла, то информация о меню будет сохраняться автоматически. Однако разработчики Delphi не зря усложняли задачу — благодаря такой сложной схеме имена файлов в списке сохраняются автоматически вместе с информацией о меню. В моем случае автомат не срабатывает, и все приходится делать вручную. Поэтому я решил не мучиться, а просто сохранять имена самостоятельно и не надеятся на свойство FileName.

Ради интереса я все же рекомендую ознакомиться с примером, который поставляется с Delphi 7. Его вы можете найти в каталоге установки Delphi 7 — Demos\ActionBands\MRU.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\Action.

Работа с «горячими» клавишами

Если нужно сделать какие-то клавиши «горячими», то есть такими, на нажатие которых реагирует активная программа, то проблема решается просто.

1. Создаем пункт меню или действие TAction. Если необходимо, то делаем его видимым на форме или в меню.
2. В свойстве ShortCut указываем сочетание клавиш.
3. Пишем обработчик события для пункта меню или действия.

Вот таким простым способом можно создать обработчик для «горячей» клавиши. Но он будет вызываться только в том случае, если активно окно программы. Если же пользователь активизирует окно другой программы и нажмет заданное сочетание клавиш, то его обработает именно эта (активная) программа, а не ваша.

Как сделать так, чтобы программа реагировала на события от «горячих» клавиш вне зависимости от того, окно какой программы активно? Для этого в Windows API есть функция регистрации «горячих» клавиш: RegisterHotkey. Рассмотрим ее работу на примере. Давайте создадим приложение, которое зарегистрирует за собой сочетание клавиш Ctrl+Alt+F1. По их нажатию окно программы будет активизироваться, располагаясь поверх остальных.

Создайте пустое приложение. В обработчике события OnShow зарегистрируем событие следующим образом:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  RegisterHotkey(Handle, 49153, MOD_CONTROL or MOD_ALT, VK_F1);
end;
```

Как видите, у функции четыре параметра:

- hWnd — указатель на окно, которому будет посылаться сообщение при нажатии «горячей» клавиши. Такой параметр есть в большинстве функций Windows API. Чаще всего он стоит первым и имеет тип HWND. Очень часто его можно

оставить нулевым, однако лучше этого не делать, указывая действительный идентификатор окна, иначе мы не получим сообщения. В данном случае указывается значение свойства Handle главной формы, где находится идентификатор окна.

- id — идентификатор сообщения. Регистрируем новое сочетание «горячих» клавиш. Для реакции на эти клавиши ОС Windows будет генерировать событие. Каждое событие имеет уникальный идентификатор. Например, событие прорисовки окна WM_PAINT имеет идентификатор 0x000F в шестнадцатеричной системе счисления или 15 в десятичной. Однако даже если вы укажите число 15, с событием WM_PAINT конфликта скорей всего не будет (правда, я не проверял), потому что это разные типы сообщений, хотя и работают они в ОС Windows схожим образом. Мы должны указать свой идентификатор. Если следовать рекомендациям Microsoft, чтобы этот идентификатор не конфликтовал с уже существующими, это должно быть шестнадцатеричное число от 0xC000 до 0xFFFF (или десятичное от 49 152 до 65 535). Я для примера выбрал 49 153.
- fsModifiers — клавиши модификаторы. К этим клавишам относятся Alt (MOD_ALT), Shift (MOD_SHIFT) и Ctrl (MOD_CONTROL). Можно использовать любое сочетание, но нас интересуют клавиши MOD_CONTROL или MOD_ALT.
- vk — виртуальная клавиша. Это любая клавиша, но необходимо указать ее виртуальный код. Нас интересует F1, а для нее виртуальным аналогом является константа VK_F1.

Если следовать хорошему тону программирования, то при выходе из программы нужно отменить регистрацию всего, что было зарегистрировано. Для этого используется функция UnregisterHotKey. По событию OnClose будет выполняться следующий код:

```
UnregisterHotKey(Handle, 49153);
```

У этой функции два параметра, которые совпадают с первыми двумя параметрами функции RegisterHotkey. Это значит, что нам нужно передать идентификатор окна и идентификатор созданного сообщения.

Сочетание клавиш зарегистрировано, и теперь наша задача заключается в том, чтобы перехватить и правильно обработать необходимое событие. Для этого в разделе private формы описываем новый метод:

```
procedure WMHotKey(var Msg:TMsg); message WM_HOTKEY;
```

Здесь объявлена процедура WMHotKey, которая реагирует на сообщения типа WM_HOTKEY. В качестве параметра в эту процедуру передается переменная Msg типа TMsg, которая в своем поле Message хранит идентификатор. Реализация обработчика совершенно элементарна:

```
procedure TForm1.WMHotKey(var Msg: TMsg);
begin
 if Msg.Message = 49153 then
  SetForegroundWindow(Handle);
 inherited;
end;
```

Наша задача проверить поле Message. Если оно равно 49 153, значит, пользователь нажал клавиши Ctrl+Alt+F1, и мы должны вывести окно программы поверх остальных окон.

Обратите внимание, что последней строкой в процедуре (inherited) мы передаем управление такому же обработчику предка, которым чаще всего будет главная форма. Если этого не сделать, то могут быть проблемы с обработкой других «горячих» клавиш или даже событий. Не забывайте вызывать предков!

Попробуйте запустить программу и посмотреть на результат. Событие OnShow должно генерироваться вне зависимости от того, какая программа активна, выводя на передний план окно созданной нами программы.

Давайте усложним задачу. Наделим ее возможностью динамического изменения «горячих» клавиш. Допустим, что вы хотите предоставить пользователю возможность самому менять «горячие» клавиши. Как это сделать? Давайте поместим на форму компоненты TEdit и THotKey (вкладка Win32 палитры компонентов). Второй компонент очень удобен для задания «горячих» клавиш. Достаточно передать ему фокус (щелкнуть или выделить), и все нажатия клавиш будут отображаться в нем так, как должны выглядеть «горячие» клавиши. У этого компонента есть еще одно достоинство — он исключает возможность ошибки.

А зачем нам нужен компонент TEdit? Дело в том, что если на форме оставить только компонент THotKey, то при активной форме программы он постоянно будет активным и любое нажатие клавиши изменит значение в THotKey. То есть компонент TEdit установлен только для того, чтобы можно было перевести на него фокус ввода с компонента THotKey.

Можете расположить компоненты как угодно, можно даже сделать подписи (рис. 2.8).

Рис. 2.8. Форма будущей программы

По событию OnShow главной формы будем вызывать метод RegisterShortCut. Этот же метод нужно вызывать по событию OnChange для компонента THotKey, но перед его вызовом нужно отменить регистрацию предыдущей клавиши:

```
procedure TForm1.HotKey1Change(Sender: TObject);
begin
 UnregisterHotKey(Handle, 49154);
 RegisterShortCut;
end;
```

В данном методе RegisterShortCut мы будем регистрировать событие, указанное пользователем (листинг 2.21).

Листинг 2.21. Регистрация события

```
procedure TForm1.RegisterShortCut;
var
 Key: Word;
 ShiftState: Word;
 State: TShiftState;
begin
  // Выбрать виртуальную клавишу
  ShortCutToKey(HotKey1.HotKey, Key, State);

  // Определение клавиш-модификаторов
  ShiftState := 0;
  if ssShift in State then
   ShiftState := ShiftState or MOD_SHIFT;
  if ssAlt in State then
   ShiftState := ShiftState or MOD_ALT;
  if ssCtrl in State then
   ShiftState := ShiftState or MOD_CONTROL;

  // Регистрация события
  RegisterHotkey(Handle, 49154, ShiftState, Key);
 end;
```

Главная проблема в том, что компонент THotKey в свойстве HotKey с помощью типа TShortCut возвращает нам полное сочетание клавиш, а для регистрации нужно отделить от «горячих» клавиш модификаторы (клавиши Ctrl, Alt и Shift). Для этого очень хорошо подходит функция ShortCutToKey. Она объявлена в модуле manu (добавьте его в раздел uses, если его еще там нет). Этой функции необходимо передать три параметра:

- сочетание клавиша в виде переменной типа TShortCut;
- переменная типа Word, в которой будет сохранена виртуальная клавиша;
- переменная типа TShiftState, в которой будут сохранены модификаторы.

Второй параметр возвращает переменную типа Word, что нас устраивает полностью. А вот с модификаторами имеется еще одна проблема. Они возвращаются в виде переменной типа TShiftState, а для регистрации «горячей» клавиши нам нужен простой тип Word. Чтобы решить проблему, приходится вручную формировать переменную ShiftState типа Word на основе полученных данных после выполнения функции ShortCutToKey.

В самой последней строке регистрируем новую «горячую» клавишу. Для нее выбран другой идентификатор — 49 154, чтобы программа могла реагировать на нее по-другому. Обработчик сообщения «горячих» клавиш в нашей программе также изменяется (листинг 2.22).

Листинг 2.22. Обработчик нажатий «горячих» клавиш

```
procedure TForm1.WMHotKey(var Msg: TMsg);
begin
 // Если событие 49153, то выводим окно поверх всех
 if Msg.Message = 49153 then
  SetForegroundWindow(Handle);
```

```
// Если событие 49154, то выводим окно поверх всех
// и изменяем положение/размер окна на случайные
if Msg.Message = 49154 then
 begin
  SetForegroundWindow(Handle);
  SetWindowPos(Handle, HWND_TOP, random(10),
    random(10), random(500), random(500), SWP_SHOWWINDOW);
 end;

 inherited;
end;
```

Я думаю, что все понятно и без дополнительных пояснений. Запустите код примера и убедитесь в его работоспособности.

Вполне логичным было бы, чтобы программа умела запоминать введенную пользователем «горячую» клавишу. Как это сделать, если свойство HotKey возвращает нам тип TShortCut, а объект TRegIniFile, применяемый для работы с реестром, не понимает этот тип? Проблема решается очень просто сохранять «горячую» клавишу нужно как число:

```
var
 RegFile:TRegIniFile;
begin
 RegFile:=TRegIniFile.Create('Software');
 RegFile.OpenKey('VR-online', true);
 RegFile.WriteInteger('TestHotKey',
     'HotKey1', HotKey1.HotKey);
 RegFile.Free;
end;
```

Читать также нужно число, но что же тогда указать в качестве значения по умолчанию? Можно указать нулевое значение, но я опытным путем выяснил, что клавиши Ctrl+Alt+F1 дают число 24 688. Как я это выяснил? Запустил программу без загрузки значения нажатого сочетания клавиш из реестра, но с сохранением этого значения. После выхода из программы посмотрел, что находится в реестре, и увидел указанное выше число.

Получается, что загрузить значение «горячей» клавиши можно следующим образом:

```
var
 RegFile:TRegIniFile;
begin
 RegFile:=TRegIniFile.Create('Software');
 RegFile.OpenKey('VR-online', true);
 HotKey1.HotKey:=RegFile.ReadInteger('TestHotKey',
   'HotKey1', 24688);
 RegFile.Free;
end;
```

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\HotKeyExamp.

Красивые меню

Не все любят действия (TAction) так, как люблю их я. Некоторые до сих пор предпочитают классический компонент TMainMenu, но ведь выглядит-то он совершенно анахронично. Это одна из причин, по которой я перешел на действия. Вторая причина заключается в том, что действия намного удобнее классических меню, хотя легко работают в связке.

Давайте заставим наше меню работать в стиле XP. Для этого я написал три универсальные процедуры, которые заставляют отображаться в современном стиле любые меню, установленные на форме, и даже выпадающие меню TPopupMenu (рис. 2.9).

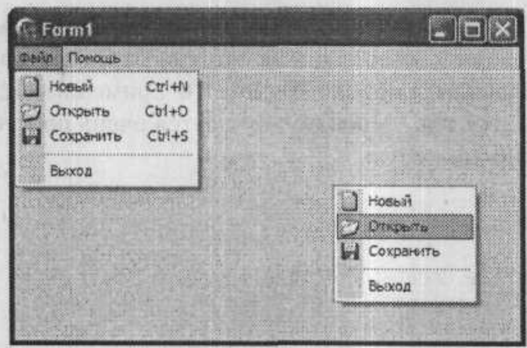


Рис. 2.9. Меню в стиле XP на основе классических компонентов

Как заставить меню выглядеть современно? Самый простой вариант — взять классическое меню, установить для каждого пункта свой обработчик события OnAdvancedDrawItem и вызывать его всякий раз, когда требуется перерисовать пункт меню. В таком обработчике можно самостоятельно нарисовать что угодно.

На первый взгляд, задача кажется сложной. Мне во времена MS-DOS приходилось рисовать вещи и посложнее. Тут вспоминается и работа с первыми версиями Delphi, когда рисунки к меню нужно было прорисовывать самостоятельно.

Итак, давайте создадим новое приложение. Поместите на форму компоненты TMainMenu, TPopupMenu и ImageList и создайте с их помощью несколько пунктов меню. Обработчики событий нас пока не особо волнуют, с ними разберемся чуть позже.

Обработчик события OnAdvancedDrawItem тоже пока незачем устанавливать, все будет делаться автоматически при старте программы. Это позволит гарантировать, что абсолютно со всеми пунктами меню будут связаны необходимые события.

Давайте для начала создадим обработчик события OnCreate и в нем выполним автоматическую установку обработчика (листинг 2.23).

Листинг 2.23. Обработчик события OnCreate

```
procedure TForm1.FormCreate(Sender: TObject);
var
 i, j:Integer;
 Menu: TMenu;
```

```
begin
 for i := 0 to ComponentCount-1 do
  begin
   if (Components[i] is TMenu) then
    begin
     Menu  := TMenu(Components[i]);
     for j := 0 to Menu.Items.Count-1 do
      SetDrawEvent(Menu.Items[j]);
    end;
  end;
end;
```

Идея кода очень проста: запускаем цикл перебора всех компонентов текущей формы, и если какой-либо компонент представляет собой меню, вызываем процедуру SetDrawEvent. Можно немного усложнить пример и вынести цикл в отдельную процедуру (этой процедуре потребуется передавать указатель на форму, компоненты которой нужно просматривать), но я этого делать не стал, дабы упростить пример.

Процедура SetDrawEvent перебирает все пункты найденного меню и устанавливает им в обработчик события OnAdvancedDrawItem процедуру DrawMenuItem:

```
procedure TForm1.SetDrawEvent(MenuItem : TMenuItem);
var
  i : integer;
begin
 MenuItem.OnAdvancedDrawItem := DrawMenuItem;
 for i := 0 to MenuItem.Count-1 do
  SetDrawEvent(MenuItem.Items[i]);
end;
```

Обратите внимание, после установки обработчика события запускается цикл перебора дочерних пунктов, и для всех найденных пунктов функция рекурсивно вызывает сама себя. Меню у программы может быть любой сложности, а рекурсия позволяет гарантировать просмотр всех пунктов.

Осталось только создать обработчик события и написать код прорисовки разных элементов меню (пунктов меню и разделителей). Обработчик должен быть объявлен в разделе private следующим образом:

```
procedure DrawMenuItem(Sender : TObject; ACanvas : TCanvas;
  ARect : TRect; State : TOwnerDrawState);
```

Код самого обработчика показан в листинге 2.24.

Листинг 2.24. Прорисовка элемента меню

```
procedure TForm1.DrawMenuItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; State: TOwnerDrawState);
var
 rtText, rtIcon: TRect;
 clBackground, clSelectedBk, clFont: TColor;
 clDisabledFont, clSeparator, clSelectedBorder: TColor;
 i: integer;
 miParent: TMenuItem;
 Menu: TMenu;
```

продолжение ⇗

Листинг 2.24 (продолжение)

```
    il: TCustomImageList;
  begin
    miParent := TMenuItem(Sender);
    Menu := miParent.Parent.GetParentMenu;

    // Задаем цвета
    clBackground := $00EFFFFF;
    clSelectedBk := $00EAC8B7;
    clSelectedBorder := $00C66931;
    clFont := clBlack;
    clDisabledFont := clGray;
    clSeparator := $00D1D1D1;
    il:=nil;
    // Определяем области картинки и текста
    rtIcon:=aRect;
    rtIcon.Right:=rtIcon.Left+20;
    rtText:=aRect;
    rtText.Left:=rtIcon.Left+20;

    // Определяем список картинок
    if ((miParent.Parent.GetParentMenu.Images <> nil) or
       (miParent.Parent.SubMenuImages <> nil)) and
       (miParent.ImageIndex <> -1) then
      if (miParent.Parent.SubMenuImages <> nil) then
       il:=miParent.Parent.SubMenuImages
      else
       il:=miParent.Parent.GetParentMenu.Images;

    // Закрашиваем цветом clBackground текстовую область меню
    aCanvas.Brush.Color := clBackground;
    aCanvas.FillRect(rtText);

    // Если это главное меню, то у него картинки и полоски для картинки нет,
    // поэтому закрашиваем его полностью цветом фона кнопки clBtnFace
    if (Menu is TMainMenu) then
        for i := 0 to miParent.GetParentMenu.Items.Count-1 do
       if (miParent.GetParentMenu.Items[i] = miParent) then
        begin
         aCanvas.Brush.Color := clBtnFace;
         aCanvas.FillRect(aRect);
         rtIcon.Right:=rtIcon.Left;
         if (miParent.ImageIndex = -1) and
            (miParent.Bitmap.Width = 0) then
          begin
           rtText  := aRect;
           Break;
          end;
        end;
   end;

    // Доступен ли выделенный компонент?
    if miParent.Enabled then
```

```
  aCanvas.Font.Color  := clFont
 else
  aCanvas.Font.Color  := clDisabledFont;

 // Если компонент выделен, то красим фон текста выделенным цветом
 if (odSelected in State) or (odHotLight in State) then
  begin
   aCanvas.Brush.Style := bsSolid;
   aCanvas.Brush.Color := clSelectedBk;
   aCanvas.Pen.Color   := clSelectedBorder;
   aCanvas.Rectangle(ARect);
  end
 // Иначе красим белым цветом
 else
  begin
   aCanvas.Brush.Color := clBtnFace;
   aCanvas.FillRect(rtIcon);
  end;

 // Если это не разделитель, выводим текст заголовка
 if not miParent.IsLine then
  begin
   SetBkMode(aCanvas.Handle, TRANSPARENT);
   if miParent.Default then
    begin
     aCanvas.Font.Color  := clGray;
     aCanvas.TextOut(rtText.Left+1, rtText.Top+1, PChar(' ' +
          miParent.Caption));
    end;

   if miParent.Enabled then
    aCanvas.Font.Color  := clFont;
   aCanvas.TextOut(rtText.Left+2, rtText.Top+2, PChar(' ' +
          miParent.Caption));
   aCanvas.TextOut(rtText.Right-2-
        aCanvas.TextWidth(ShortCutToText(miParent.ShortCut) + ' '),
        rtText.Top+2, PChar(ShortCutToText(miParent.ShortCut) + ' '));

   // Если элементу назначена картинка, то отображаем ее
   if (il <> nil) and (miParent.ImageIndex>=0) then
    begin
     ImageList_DrawEx(il.Handle, miParent.ImageIndex, ACanvas.Handle,
         rtIcon.Left+2, rtIcon.Top+2,
         0, 0, clNone, clNone, ILD_Transparent)
    end;
  end
 else
  begin
   aCanvas.Pen.Color := clSeparator;
   aCanvas.MoveTo(ARect.Left + 10, rtText.Top +
         Round((rtText.Bottom - rtText.Top) / 2));
   aCanvas.LineTo(ARect.Right - 2, rtText.Top +
         Round((rtText.Bottom - rtText.Top) / 2))
  end;
end;
```

Тут достаточно много интересных нюансов, поэтому давайте их подробно рассмотрим. В качестве параметра Sender передается указатель на элемент меню, который нужно прорисовать. Эта переменная имеет тип TObject, но так как там точно находится элемент типа TMenuItem, сохраняем его для удобства в переменной этого типа:

```
miParent := TMenuItem(Sender);
```

Во время прорисовки нужно знать указатель на само меню (TMainMenu или TPopupMenu):

```
Menu := miParent.Parent.GetParentMenu;
```

Теперь заполняются переменные цветов, которые будут использоваться во время рисования, а цвета я подобрал в стиле XP:

```
clBackground := $00EFFFFF;
clSelectedBk := $00EAC8B7;
clSelectedBorder := $00C66931;
clFont := clBlack;
clDisabledFont := clGray;
clSeparator := $00D1D1D1;
```

Все эти переменные можно было бы превратить в константы, если вы не собираетесь изменять цветовую гамму меню, а можно вынести в отдельный класс, чтобы можно было изменять гамму даже во время выполнения программы.

Теперь определяем размеры области изображения и текста. Если вы посмотрите на меню XP, то слева, где рисуется изображение, цвет фона немного темнее (серый), а под текстом фон белый. Общий размер и положение пункта меню находится в переменной ARect, которую мы получаем в качестве параметра. Для хранения области изображения будем использовать структуру rtIcon, а для текста — структуру rtText. В обе переменные на начальном этапе заносится значение ARect, то есть размер всего пункта меню.

В качестве размера изображения используем стандартное значение 16×16, а серую полосу сделаем шириной в 20 пикселов. Это значит, что в переменной rtIcon нужно правую позицию установить равной левой позиции плюс 20. Таким образом, ширина области, описываемой структурой rtIcon, будет равна тому самому значению в 20 пикселов.

В области текста (rtText) нужно сдвинуть левую позицию на 20 пикселов:

```
rtIcon:=aRect;
rtIcon.Right:=rtIcon.Left+20;
rtText:=aRect;
rtText.Left:=rtIcon.Left+20;
```

Получается, если текущий пункт имеет ширину 100 пикселов, то первые 20 резервируются для рисования значка на сером фоне, а остальные 80 остаются под текстом меню на белом фоне.

Список изображений ImageList может быть назначен компоненту TMainMenu в целом или конкретной ветке этого меню в свойстве SubMenuImages. Мы должны определить, есть ли изображения у данной ветки. Если нет, то нужно взять родительское изображение:

```
if ((miParent.Parent.GetParentMenu.Images <> nil) or
   (miParent.Parent.SubMenuImages <> nil)) and
```

```
  (miParent.ImageIndex <> -1) then
  if (miParent.Parent.SubMenuImages <> nil) then
    il:=miParent.Parent.SubMenuImages
  else
    il:=miParent.Parent.GetParentMenu.Images;
```

Указатель на компонент ImageList сохраняется в переменной il типа TCustomImageList.

Далее начинаем красить. Тут ничего сложного и оригинального нет, и все должно быть понятно по комментариям. Самое интересное кроется в рисовании изображения. Список изображений известен, он хранится в переменной il, а индекс изображения легко узнать с помощью свойства miParent.ImageIndex. Первое, что приходит в голову, — получить изображение методом GetBitmap. Это простое решение, но только на первый взгляд. Как такое изображение выводить в меню и при этом учитывать прозрачность? Вот тут и возникнут проблемы.

У компонента ImageList уже есть готовая функция для вывода изображения с учетом прозрачности (точнее, в WinAPI она есть), ее я нашел в исходных кодах компонентов. Там же я по примерам понял, как ее использовать. Итак, нам понадобится функция ImageList_DrawEx, которая объявлена в модуле CommCtrl. Этой функции нужно передать следующие параметры:

- указатель на компонент ImageList, изображение которого нужно нарисовать;
- индекс изображения;
- указатель на контекст устройства (Canvas), на котором нужно рисовать;
- левую позицию на контексте устройства, куда нужно выводить изображение;
- правую позицию на контексте;
- ширину (если ноль, то выводить все);
- высоту (если ноль, то выводить все);
- цвет переднего плана;
- цвет фона;
- флаги рисования (в нашем случае стоит флаг ILD_Transparent, что означает необходимость прозрачной прорисовки).

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\MenuXP.

На рисунке 2.10 можно увидеть созданное с помощью приведенного кода меню. Черно-белый рисунок скорее всего не сможет передать всех нюансов, поэтому советую посмотреть пример в каталоге Source\ch02\MenuXP2.

Рис. 2.10. Меню в стиле XP

Рисовать самостоятельно можно многие стандартные компоненты. ОС дает нам только функциональность, события и отображение по умолчанию. Но изменить внешний вид вы можете самостоятельно. Не стесняйтесь рисовать, это не так уж сложно. Я считаю, что самое трудное — учесть все возможные состояния. Например, у пунктов меню есть состояния *выделен* (Checked) и *доступен* (Enabled), в меню имеются разделители, пунктам меню могут быть назначены изображения и клавиши быстрого вызова. В нашем примере состояние Checked не прорисовывается. Подобную прорисовку я оставил вам в качестве домашнего задания.

Кроме того, в качестве домашнего задания могу предложить создать меню, в которых вместо текста прорисовывается цветной прямоугольник. Если в вашей программе есть код для работы с цветом, то неплохо было бы добавить в него фрагмент для быстрого выбора основных цветов. Окрашивать в выбранный цвет можно только квадратный значок с изображением или весь пункт меню.

Коллекции

Если вы уже знакомы с моими работами, то должны знать, что для динамического хранения списка объектов я люблю использовать компонент TList. Например, если бы при динамическом создании изображений (см. раздел «Сохранение данных проекта») я захотел бы сохранить указатель на изображения с дополнительной информацией в списке, то таким списком был бы компонент TList. В главе 4 мы будем рассматривать примеры динамического сохранения списков объектов и тоже будем использовать TList. Почему? Это только дело привычки, ведь в Delphi есть более эффективное средство хранения списков объектов — коллекции.

Для хранения динамического массива объектов наилучшим вариантом является использование компонента TCollection. Он может хранить объекты заранее определенного типа и при добавлении элемента в коллекцию автоматически создавать новый объект. Чтобы лучше понять механизм работы коллекций, рассмотрим реальный пример.

Допустим, нам нужно хранить в памяти динамический список работников, который будет включать в себя такую информацию, как фамилия, имя и дата рождения, пол и т. д. Чтобы не усложнять пример, ограничимся только этими данными.

Для начала создадим объект, который будет хранить все необходимые свойства. При этом объект должен быть наследником от TCollectionItem:

```
TPersonInfo = class (TCollectionItem)
 public
  Name: String;
  SurName: String;
  Pol : char;
  Age: Integer;
end;
```

Наследование от TCollectionItem является обязательным условием. Код хранения объектов такого типа с помощью коллекций показан в листинге 2.25.

Листинг 2.25. Хранение объектов в коллекции

```
procedure TForm1.bnTestCollectionClick(Sender: TObject);
var
 pi: TPersonInfo;
 i:Integer;
begin
 PersonCollection:=TCollection.Create(TPersonInfo);

 // Добавляем один элемент
 pi:=TPersonInfo(PersonCollection.Add());
 pi.Name:='Михаил';
 pi.SurName:='Фленов';
 pi.Pol:='M';
 pi.Age:=29;

 // Добавим еще элемент, но заполнять будем через with,
 // чтобы не заводить отдельную переменную
 with TPersonInfo(PersonCollection.Add()) do
  begin
   Name:='Ирина';
   SurName:='Смирнова';
   Pol:='Ж';
   Age:=24;
  end;

 // Запускаем цикл вывода информации из объектов
 for i:=0 to PersonCollection.Count-1 do
  begin
   pi:=TPersonInfo(PersonCollection.Items[i]);
   Memo1.Lines.Add('Фамилия: '+pi.SurName);
   Memo1.Lines.Add('Имя     : '+pi.Name);
   Memo1.Lines.Add('Пол     : '+pi.Pol);
   Memo1.Lines.Add('Возраст: '+IntToStr(pi.Age));
   Memo1.Lines.Add('=========================');
  end;

 PersonCollection.Free;
end;
```

Для создания новой коллекции используется метод `Create`, который вызывает конструктор объекта. В качестве параметра ему необходимо передать класс сохраняемого в коллекции объекта. В нашем случае — это `TPersonInfo`.

Для добавления нового элемента в коллекцию используется метод `Add`. Ему не надо передавать никаких параметров, метод сам создает новый объект, добавляет его в коллекцию, а в качестве результата возвращает указатель на созданный объект. В нашем случае, результат сохраняется в переменной `pi` типа `TPersonInfo`:

```
pi:=TPersonInfo(PersonCollection.Add());
```

Метод `Add` возвращает тип `TCollectionItem`, и, чтобы присвоить этот результат переменной `pi`, результат приводится к типу `TPersonInfo`. Как коллекция «узнает», какой тип объекта нужно создать? Помните, при создании коллекции мы передали конструктору тип `TPersonInfo`? Благодаря этому все работает как надо и без нашего вмешательства.

В данном примере мы создаем два элемента коллекции и заполняем их данными. После этого запускается цикл, в котором перебираются все элементы и на экран выводится информация об объектах.

Как видите, все проще и в чем-то даже удобнее, чем при использовании компонента `TList`. Напоследок вкратце рассмотрим свойства и методы коллекций. Начнем со свойств:

- `Count` — возвращает количество элементов в коллекции (помните, что элементы нумеруются с нуля);
- `Items` — список элементов в коллекции; для доступа к определенному элементу по индексу используется запись:

  `Items[`*`индекс`*`]`
- `NextID` — уникальный идентификатор, который присваивается новому элементу при добавлении его в коллекцию (каждый элемент коллекции обладает уникальным идентификатором, и эти идентификаторы можно использовать в своей программе).

Методов у объекта побольше, чем свойств:

- `Add` — создать новый элемент коллекции и вернуть на него указатель;
- `Assing` — скопировать элементы из указанной коллекции (параметром метода является коллекция, элементы которой нужно скопировать);
- `EndUpdate` — остановить обновление экрана;
- `BeginUpdate` — восстановить возможность обновления экрана;
- `Changed` — возвращает `true`, если коллекция или один из ее элементов изменен;
- `Clear` — удалить все элементы коллекции;
- `Create` — создать коллекцию (в качестве параметра нужно указать тип объекта, элементы которого будут храниться в коллекции);
- `Delete` — удалить один элемент из коллекции (в качестве параметра нужно указать индекс удаляемого элемента);
- `FindItemID` — найти элемент коллекции по его идентификатору;
- `Insert` — создать новый элемент и вставить его в коллекцию под определенным индексом (например, если вы хотите, чтобы новый элемент был первым, и чтобы вы могли обращаться к нему как `Items[0]`, используйте метод `Insert`, указав в качестве параметра индекс 0).

Это основные методы и свойства, которые могут понадобиться вам во время разработки собственных приложений.

Помимо самой коллекции необходимо хорошо разбираться в ее элементах, которые являются наследниками класса `TCollectionItem`, а значит, наследуют его свойства и методы. Давайте рассмотрим основные свойства и методы:

- `DisplayName` — имя, которое должно отображаться в редакторе коллекции;
- `ID` — уникальный идентификатор, который автоматически присваивается элементу во время его создания;
- `Index` — возвращает индекс элемента в коллекции;

- Changed — этот метод возвращает true, если элемент был изменен.

Хотя коллекции предлагают весьма удобный способ хранения объектов (структур данных, которые можно реализовать в виде объекта, наследника от TCollectionItem), я все же привык к структурам и объекту TList, поэтому в данной книге буду использовать именно этот метод. Но это не значит, что коллекции плохи, наоборот, они хороши, просто тут действует привычка.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\MenuXP.

Работа с сеткой StringGrid

При разработке приложений иногда удобно использовать сетку StringGrid. Если вы сталкивались с этим компонентом, то наверно заметили, что у него довольно много ограничений, а вот возможностей недостаточно. Большинство необходимых возможностей приходится реализовывать самостоятельно путем рисования ячеек. Чтобы вам было проще, в этом разделе мы рассмотрим некоторые проблемы, с которыми я сталкивался в своей работе.

Секреты и проблемы StringGrid

Допустим, мы хотим написать программу учета расходов. Для этого нам понадобится таблица из четырех колонок, в которых будут указаны название товара, его количество, цена и общая сумма расходов. Поскольку у нас четыре колонки, свойство ColCount устанавливаем в 4.

При старте в сетке должна быть только одна строка (свойство RowCount равно 1) с заголовками колонок, причем эта строка должна быть фиксированной (ее фон должен иметь цвет фона кнопки). Первая колонка также должна быть фиксированной. Вот тут возникает первая проблема — количество фиксированных строк должно быть меньше общего количества строк, так же как количество фиксированных колонок должно быть меньше общего количества колонок. В нашем случае у нас есть только одна строка, и ее нельзя сделать фиксированной, пока количество строк не превысит 1.

Придется рисовать фиксированные строки и колонки самостоятельно, поэтому свойства FixedCols и FixedRows делаем равными нулю. После этого создаем обработчик события OnDrawCell для компонента StringGrid и пишем в нем следующий код:

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject;
  ACol, ARow: Integer;
  Rect: TRect; State: TGridDrawState);
begin
 if (ACol=0) or (ARow=0) then
  StringGrid1.Canvas.Brush.Color:=clBtnFace;

 StringGrid1.Canvas.FillRect(Rect);
 StringGrid1.Canvas.TextOut(Rect.Left+2, Rect.Top+1,
```

```
    StringGrid1.Cells[ACol, ARow]);
end;
```

Код достаточно прост. Если ячейка, которую нужно прорисовать, находится в нулевой строке (параметр ARow равен 0) или нулевой колонке (параметр ACol равен 0), то цвет кисти для холста StringGrid1 делаем равным clBtnFace, то есть соответствующим цвету фона кнопки. После этого закрашиваем ячейку текущим цветом:

```
StringGrid1.Canvas.FillRect(Rect);
```

Так как мы всю ячейку закрасили, нужно самостоятельно прорисовать текст ячейки, что мы и делаем в последней строке:

```
StringGrid1.Canvas.TextOut(Rect.Left+2, Rect.Top+1,
    StringGrid1.Cells[ACol, ARow]);
```

Усложним задачу. Пусть значения в последней колонке нашей таблицы (это общая стоимость товара) получаются расчетным путем. Такую колонку надо как-то выделить на фоне других, чтобы пользователь мог видеть, что ее нельзя редактировать напрямую. Окрасим фон этой колонки (кроме первой строки) в красный цвет, а чтобы текст на этом фоне был читабельным, в качестве цвета шрифта выберем белый.

Но и это еще не все. В нашем примере подсчет производится только по горизонтали, а что если делать это и по вертикали, чтобы внизу была строка с итоговыми данными? Ее окрасим в зеленый цвет. Для этого изменим свойство RowCount на 2, тогда при старте в окне программе будут видны две строки — одна с заголовками колонок, другая с итоговыми данными.

Чтобы выполнить раскраску колонки и строки с итоговыми данными, изменяем обработчик события прорисовки ячейки, как показано в листинге 2.26.

Листинг 2.26. Прорисовка ячейки

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject;
  ACol, ARow: Integer;
  Rect: TRect; State: TGridDrawState);
begin
 // Устанавливаем цвет, если текущая колонка или строка равна 0
  if (ACol=0) or (ARow=0) then
   StringGrid1.Canvas.Brush.Color:=clBtnFace;

 // Устанавливаем цвета, если текущая колонка последняя
  if (ACol=3) and (ARow<>0) then
   begin
    StringGrid1.Canvas.Brush.Color:=clRed;
    StringGrid1.Canvas.Font.Color:=clWhite;
   end;

 // Устанавливаем цвета, если текущая строка последняя
  if (ARow=StringGrid1.RowCount-1) then
   begin
    StringGrid1.Canvas.Brush.Color:=clGreen;
    StringGrid1.Canvas.Font.Color:=clWhite;
   end;
```

```
  // Рисуем
  StringGrid1.Canvas.FillRect(Rect);
  StringGrid1.Canvas.TextOut(Rect.Left+2, Rect.Top+1,
    StringGrid1.Cells[ACol, ARow]);
end;
```

Как теперь организовать добавление строк? Ведь очередная строка должна вставляться перед последней, а класс `StringGrid` не позволяет делать подобного. Итак, на форму добавим кнопку вставки строки, по щелчку на которой будет выполняться код из листинга 2.27.

Листинг 2.27. Вставка строк

```
procedure TForm1.acAddExecute(Sender: TObject);
var
 Str:String;
Begin
 Str:='Новая строка';

 // Запрашиваем имя строки (название товара)
 if not InputQuery('Добавление строки',
    'Введите название новой колонки', Str) then
  exit;

 // Добавляем товар
 StringGrid1.RowCount:=StringGrid1.RowCount+1;
 StringGrid1.Cells[0, StringGrid1.RowCount-1]:='Сумма';
 StringGrid1.Cells[0, StringGrid1.RowCount-2]:=Str;
 StringGrid1.Cells[1, StringGrid1.RowCount-2]:='0';
 StringGrid1.Cells[2, StringGrid1.RowCount-2]:='0';
 StringGrid1.Selection:=TGridRect(Rect(1, StringGrid1.RowCount-2,
    1, StringGrid1.RowCount-2));
end;
```

Самое интересное происходит при добавлении данных о товаре. Сначала мы увеличиваем количество строк на единицу. Потом название последней строки меняем на «Сумма», а название предпоследней строки — на название добавляемого товара. Не забывайте, что строки нумеруются с нуля, и чтобы получить доступ к последней строке нужно вычесть единицу из значения свойства `RowCount` компонента `StringGrid`. Для предпоследней строки нужно вычесть число 2.

Все колонки новой строки обнуляются, а в последней строке мы выделяем колонку с количеством товара (под номером 1):

```
StringGrid1.Selection:=TGridRect(Rect(1,
  StringGrid1.RowCount-2, 1, StringGrid1.RowCount-2));
```

За выделение отвечает свойство `Selection`, которое имеет тип `TGridRect`. Давайте рассмотрим его поподробнее, поскольку это свойство почему-то очень плохо документировано. Тип `TGridRect` идентичен типу `TRect`. Посмотрите на объявление этого типа, которое я взял из исходного кода модуля **grid.pas**:

```
TGridRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Longint);
    1: (TopLeft, BottomRight: TGridCoord);
end;
```

Я даже не знаю, зачем нужно было вводить новый тип, если можно использовать тип TRect.

Итак, как работать с этим свойством? Одновременно может быть выделено несколько ячеек. Координаты первой выделенной колонки определяются свойством StringGrid1.Selection.Left, последней — свойством StringGrid1.Selection.Right. Подобным же образом можно определить координаты первой выделенной строки — StringGrid1.Selection.Top. Догадайтесь с трех раз, как отыскать последнюю выделенную строку?

Все эти свойства можно изменять, чтобы контролировать выделенную область. В представленном примере выделяем ячейки, заданные областью: 1, StringGrid1.RowCount-2, 1, StringGrid1.RowCount-2. Эти четыре координаты описывают только одну ячейку.

Чтобы быстро получить тип TGridRect, не заполняя каждое свойство в отдельности, можно использовать функцию Rect, как в представленном ранее примере. Функция выглядит следующим образом:

```
function Rect(
  ALeft: Integer,
  ATop: Integer,
  ARight: Integer,
  ABottom: Integer
): TRect;
```

Этой функции передаются четыре координаты, и в конечном тоге мы получаем результат в виде типа TRect. Так как этот тип схож с TGridRect, можно выполнить приведение типов:

```
TGridRect(Rect(параметры))
```

Попробуйте теперь запустить программу и добавить несколько строк. Убедитесь, что все работает корректно и все строки/колонки нарисованы нужным цветом (рис. 2.11).

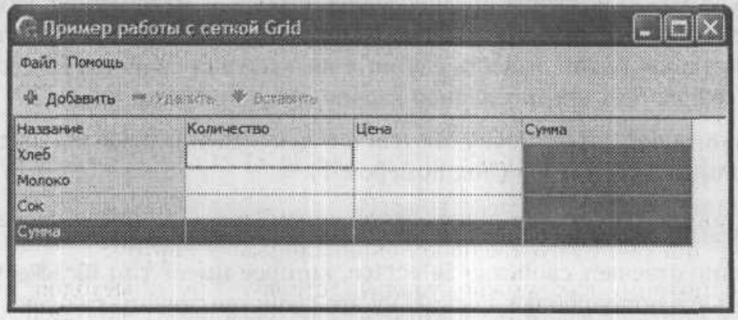


Рис. 2.11. Результат работы первой версии программы

Если вы протестируете пример, то заметите, что можно выделять любые ячейки. А ведь фиксированные колонки и колонки с итоговыми данными незачем выделять и тем более редактировать. Чтобы исправить этот недостаток, создадим обработчик события OnSelectCell. В качестве последнего параметра в этот обработ-

чик будем передавать параметр CanSelect. Если ему присвоить значение false, то выделение станет невозможным. Давайте напишем такой код, в котором, если выделена фиксированная ячейка или ячейка итогов, переменной будет присваиваться значение false:

```
procedure TForm1.StringGrid1SelectCell(Sender: TObject;
  ACol, ARow: Integer; var CanSelect: Boolean);
begin
 if (ARow=0) or (ACol=0) or (ACol=3) or
    (ARow=StringGrid1.RowCount-1) then
  CanSelect:=false;
end;
```

Теперь добавим возможность расчета итогов. Для этого я создал обработчик события OnSelEditText и в нем вызываю процедуру Calculate. Саму процедуру можно увидеть в листинге 2.28.

Листинг 2.28. Расчет итогов

```
procedure TForm1.Calculate;
var
 i:Integer;
 iNumber, iSumm:Integer;
begin
 iNumber:=0;
 iSumm:=0;

 // Запускаем цикл
 for i:=1 to StringGrid1.RowCount-2 do
  begin
   // Расчет суммы расходов текущей строки
   StringGrid1.Cells[3, i]:=IntToStr(
     StrToIntDef(StringGrid1.Cells[1, i], 0)*
     StrToIntDef(StringGrid1.Cells[2, i], 0)
     );

   // Накапливаем сумму по колонке количества и по колонке суммы
   iNumber:=iNumber+StrToIntDef(StringGrid1.Cells[1, i],0);
   iSumm:=iSumm+StrToInt(StringGrid1.Cells[3, i]);
  end;

 // Выводим суммы по вертикали
 StringGrid1.Cells[1, StringGrid1.RowCount-1]:=IntToStr(iNumber);
 StringGrid1.Cells[3, StringGrid1.RowCount-1]:=IntToStr(iSumm);
end;
```

Теперь окно программы выглядит законченным (рис. 2.12). Для удобства и красоты я выделил текст заголовков колонок жирным шрифтом.

Теперь посмотрим, как можно удалить строку. Среди методов компонента TStringGrid, описанных в файле помощи, указаны методы DeleteRow и DeleteCol, но использовать их вам не удастся. Дело в том, что они защищены и, чтобы получить к ним доступ, надо действовать так, как описано в разделе «Доступ к защищенным свойствам и методам». Для этого создаем в модуле новый объект типа TStringGrid:

```
TMyStringGrid=class(TStringGrid)
end;
```

Рис. 2.12. Результат работы окончательной версии программы

После этого методы `DeleteRow` и `DeleteCol` становятся доступными. Вот так может произойти удаление выделенной строки:

```
procedure TForm1.acDelExecute(Sender: TObject);
begin
 TMyStringGrid(StringGrid1).DeleteRow(
    StringGrid1.Selection.Top);
 Calculate;
end;
```

Удаляется верхняя выделенная строка (`StringGrid1.Selection.Top`). После удаления вызываем метод для расчета итогов.

Как видите, даже в достаточно простом примере применения `StringGrid` компонента мы столкнулись с массой проблем.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\Grid.

Сохранение и загрузка данных

Компонент `StringGrid` не имеет методов для загрузки и восстановления данных, поэтому их реализация ложится на наши плечи. Я думаю, что разработчики Borland просто не нашли универсального способа решения проблемы. Действительно, ведь мы не знаем, какие данные могут быть расположены в сетке и сколько их будет.

В нашей сетке всего 4 колонки, и если взять это количество за правило, то реализовать сохранение будет несложно. Например, можно сохранить данные все в том же формате XML или даже в базе данных. При фиксированном количестве колонок можно найти сотню вариантов решения проблемы.

Однако не будем искать легких путей и сделаем универсальную программу загрузки и сохранения данных. Мой метод основан на создании таблицы в обычном текстовом файле, причем каждая строка этой таблицы представляет собой

строку сетки StringGrid. Все значения для колонок записываются в строках этой таблицы и должны каким-то образом разделяться. Для этого можно использовать любой неалфавитный и не используемый в сетке символ, например символ табуляции (он имеет код #9).

Возможный вариант кода для сохранения данных можно увидеть в листинге 2.29.

Листинг 2.29. Сохранение данных сетки StringGrid

```
procedure TForm1.acSaveExecute(Sender: TObject);
var
 sl:TStringList;
 i, j:Integer;
 Str:String;
Begin
  // Создание окна диалога
  with TSaveDialog.Create(Owner) do
   begin
    // Выбор файла
    DefaultExt:='.sav';
    Filter:='Файлы проекта|*.sav';
    if not Execute then exit;

    // Сохранение содержимого сетки
    sl:=TStringList.Create;

    // Первой строкой сохраняем общее количество строк и колонок в сетке
    sl.Add(IntToStr(StringGrid1.RowCount)+#9+
          IntToStr(StringGrid1.ColCount));

    // Запускаем цикл перебора всех строк сетки
    for i:=0 to StringGrid1.RowCount-1 do
     begin
      Str:='';

      // Перебираем колонки текущей строки и формируем сохраняемую строку
      for j:=0 to StringGrid1.ColCount-1 do
         Str:=Str+StringGrid1.Cells[j, i]+#9;

      // Сохраняем строку
      sl.Add(Str);
     end;

    // Сохраняем содержимое списка строк в файле
    sl.SaveToFile(FileName);
   end;
 sl.Free;
end;
```

Тут есть несколько интересных решений. Во-первых, на форме у нас нет компонентов TSaveDialog и TOpenDialog, с помощью которых можно было бы вывести стандартное окно диалога выбора файлов. Эти компоненты создаются динамически в коде. В первой строке кода мы вызываем конструктор для создания компонента TSaveDialog. Создание происходит в операторе with, а значит, последующий

код между ключевыми словами begin и end будет выполняться вместе с созданным объектом.

Прежде чем вывести на экран окно диалога, мы изменяем свойства DefaultExt и Filter. Первому из свойств назначаем расширение, присваиваемое по умолчанию (.sav). Если пользователь укажет имя файла, но не укажет расширение, то файлу будет присвоено это расширение. В свойстве Filter задается фильтр отображения файлов. Фильтр выглядит как строка и должен иметь следующий вид:

```
текст|расширение
```

Текст — это то, что пользователь видит в окне диалога в списке выбора типа отображаемых файлов. После символа вертикальной черты указывается расширение или список расширений через точку с запятой. В окне будут отображаться файлы только с этими расширениями.

Вот теперь мы готовы отобразить окно диалога выбора файла с помощью метода Execute. Если файл не выбран, выполняется выход из процедуры. Если файл выбран, выполняется последующий код сохранения.

Для сохранения текстовых данных я больше всего люблю использовать список строк (объект TStringList). Этот список удобно сформировать в памяти объекта и потом сохранить в файле одной строкой кода.

Давайте договоримся, что самая первая строка будет использоваться для хранения параметров. В нашем случае к параметрам относится размерность, то есть количество строк и колонок. Именно эти значения добавляются в список строк первыми.

После этого запускаем цикл перебора всех строк сетки. В цикле выполняется еще один цикл, который перебирает все колонки. Значение каждой колонки прибавляется к переменной Str и добавляется разделитель (символ табуляции, который имеет код #9). После учета всех колонок в переменной Str, полученное значение прибавляется к списку строк.

Сформированный список всех строк сохраняется в выбранном пользователем файле с помощью метода SaveToFile.

Запустите программу, создайте несколько строк и попробуйте сохранить результат в файле. После этого откройте полученный файл в текстовом редакторе (например, в блокноте). Вы должны увидеть нечто подобное:

```
6        4
Название  Количество  Цена  Сумма
Хлеб      20          8     160
Молоко    4           18    72
Сок       7           23    161
Вода      5           11    55
Сумма     36                448
```

В первой строке находятся два числа, разделенных табуляцией. Эти числа отражают количество строк и колонок соответственно. Убедитесь, что числа действительно соответствуют размерности таблицы. Начиная со второй строки мы видим саму таблицу.

Как теперь выполнить загрузку этих данных? Мой код загрузки можно увидеть в листинге 2.30.

Листинг 2.30. Загрузка данных сетки из файла

```
procedure TForm1.acOpenExecute(Sender: TObject);
var
 sl:TStringList;
 i, j:Integer;
 Str, CellText:String;
Begin
 // Динамически создаем окно выбора файла
 with TOpenDialog.Create(Owner) do
  begin
   // Устанавливаем параметры окна и отображаем
   DefaultExt:='.sav';
   Filter:='Файлы проекта|*.sav';
   if not Execute then exit;

   // Создаем список строк и загружаем в него файл
   sl:=TStringList.Create;
   sl.LoadFromFile(FileName);

   // Определяем количество строк
   Str:=sl[0];
   i:=StrToInt(copy(Str, 1, pos(#9, Str)-1));
   StringGrid1.RowCount:=i;
   Delete(Str, 1, pos(#9, Str));

   // Определяем количество колонок
   StringGrid1.ColCount:= StrToInt(Str);

   // Загружаем ячейки
   for i:=1 to sl.Count-1 do
    begin
     Str:=sl[i];
     j:=0;
     while Length(Str)>0 do
      begin
       CellText:=copy(Str, 1, pos(#9, Str)-1);
       StringGrid1.Cells[j, i-1]:=CellText;
       Delete(Str, 1, pos(#9, Str));
       inc(j);
      end;
    end;

  end;
 sl.Free;
end;
```

В данном примере мы также создаем окно выбора файла динамически, устанавливаем его параметры и отображаем. После этого создается список, и в него загружаются данные из выбранного файла.

Далее выполняется непосредственно сама загрузка. В самой первой строке находится количество строк и колонок. Сохраняем значение этой строки в перемен-

ной Str, чтобы удобно было разделить ее на части. Чтобы определить количество строк, выделяем из строки Str символы с первого символа до символа табуляции и преобразуем их в число следующим образом:

```
i:=StrToInt(copy(Str, 1, pos(#9, Str)-1));
```

Теперь в переменной i находится количество строк. Чтобы определить количество колонок, достаточно удалить из строки Str все символы до табуляции, то есть информацию о количестве строк:

```
Delete(Str, 1, pos(#9, Str));
```

После этого в переменной Str у нас остается только количество колонок.

Далее начинаем перебор всех строк. Каждая строка сохраняется в переменной Str для последующего разбора. Разбор состоит из цикла while, который выполняется, пока строка не становится пустой. Внутри этого цикла мы копируем в переменную CellText символы, начиная с первого и заканчивая первым найденным с помощью функции Pos символом табуляции. Скопированный текст присваивается соответствующей ячейке, а потом удаляется из переменной Str, чтобы на следующем этапе копировать текст очередной ячейки.

Как видите, универсальный метод может существовать, но для этого должны соблюдаться два условия.

- В содержимом ячеек нельзя использовать многострочный текст, иначе он нарушит структуру текстового файла, и при загрузке данные окажутся не в том виде, в котором мы ожидали.
- Нельзя использовать в ячейках символ табуляции. Мы его применяем в файле в качестве разделителя. Если в ячейках должен присутствовать символ табуляции, то при сохранении придется задействовать какой-нибудь другой неалфавитный символ.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch02\Grid1.

Глава 3

Базы данных

Почему тему баз данных считают сложной? Я думаю, что это связано с необходимостью знать не только программирование в Delphi, но и язык запросов SQL (Structured Query Language — структурированный язык запросов), а также иметь навыки администрирования баз данных. Это пугает не только начинающих программистов, но и опытных. Однако поверьте мне, ничего сложного во всех этих вопросах нет.

В данной главе мы не будем учиться работать с базами данных. Об этом написано уже немало книг. Но мы научимся эффективно ими управлять, рассмотрим алгоритмы решения некоторых особо часто встречающихся проблем. Во всех моих программах, если они не являлись сугубо личными проектами, а писались для заказчика, всегда использовались базы данных. Если работать программистом на какой-нибудь фирме, то 90 % создаваемого кода будет связано именно с базами данных.

Если вы хорошо изучите эту тему, то сможете легко создавать складские, бухгалтерские и другие программы для баз данных. И когда-нибудь вы несомненно обрадуете свое начальство, предложив более удобную программу, чем универсальный продукт.

Любые универсальные продукты для предприятий, бухгалтерские, складские и т. д. (так называемые коробочные, то есть те, дистрибутивные диски для которых продаются в коробках вместе с многочисленными техническими описаниями и руководствами), предназначены только для небольших фирм и офисов, где нет возможности содержать программистов. Крупным предприятиям лучше отказаться от использования таких продуктов, потому что за универсализм в конкретных условиях приходится расплачиваться удобством и быстродействием. Да, разработка собственного программного продукта требует определенных трудовых затрат, но зато позволяет получить то, что нужно именно вам и в отношении удобства, и в отношении быстродействия.

Для описываемых в этой главе примеров я выбрал сервер MS SQL Server и компоненты ADO (Active Data Objects — активные объекты данных). Этот выбор я сделал потому, что SQL Server становиться одним из самых распространенных серверов в России, поскольку обладает достаточно развитыми возможностями. Благодаря тому, что в Delphi все компоненты доступа к базам данных схожи в использовании (имеют одинаковые свойства, методы и события), перенос кода на другую базу данных сложностей обычно не вызывает.

Чем отличаются предлагаемые мною примеры от примеров из других книг по программированию и базам данных? Тем, что мои примеры взяты из жизни и поэтому могут пригодиться при разработке реальных приложений.

Я надеюсь, что вы уже знаете основы программирования баз данных, поэтому описание простых операций будем опускать, а основной упор делать на логике и алгоритмах.

ПРИМЕЧАНИЕ

Я не пытался включить в одну главу книги все, что знаю о базах данных; дополнительную информацию вы найдете на компакт-диске в каталоге /doc/database. В большей степени эта информация относится к MS SQL Server, но многие приводимые там рекомендации пригодятся всем программистам. Если вы не имеете опыта программирования баз данных, рекомендую прочитать мою книгу «Библия Delphi», вышедшую в издательстве БХВ в 2004 г.

СОВЕТ

Для усвоения материала этой главы вам необходимо знать не только основы программирования баз данных, но и язык запросов SQL, который является стандартом доступа к данным. Если вы этот язык не знаете или знаете плохо, то некоторые вопросы будут для вас непонятными. Чтобы мы лучше понимали друг друга, я выложил на компакт-диск в каталог doc/sql документ, с помощью которого вы можете познакомиться с основами языка SQL.

Постановка задачи

С чего начинается любой проект? Конечно же, с постановки задачи. Мой опыт работы с конечными пользователями показывает, что они не всегда могут правильно поставить задачу, поэтому приходится это делать самому. Сначала мы рассмотрим только основы, а потом будем постепенно усложнять задачу.

Что мы будем писать? Я решил выбрать в качестве примера программу управления клиентами. Эта тема стала популярной среди бизнесменов, и программы такого типа стоят хороших денег. Прочитав эту главу, вы поймете, что в управлении клиентами ничего сложного нет, и, немного изменив мой пример, сможете создать собственную систему. Для этого достаточно добавить некоторые специфичные для ваших задач функции.

Итак, создадим базу данных клиентов фирмы, в которой будем хранить название фирмы, адрес, информацию о контактных лицах, телефоны. Пока ограничимся этими данными.

Самое главное при создании базы данных — правильно ее спроектировать. От этого зависит будущее программы. Если удастся все предусмотреть еще на этапе

проектирования, то снижается вероятность того, что программу придется переписывать «с нуля».

Но прежде чем разбираться с проектированием, я приведу правила, которым следую при именовании объектов базы данных.

- Имена таблиц начинаются с префикса `tb`, затем следует имя таблицы, которое четко отражает смысл хранящейся информации.
- Имена полей начинаются с префикса, отражающего тип поля. Рассмотрим основные префиксы, которые использую я и большинство других программистов. Поля других типов используются не так часто, да и на основе описанного вы легко сможете придумать для себя правила именования любых полей. А я буду использовать только следующие префиксы:
  - `vc` — тип `varchar`;
  - `d` — тип `datetime`;
  - `i` — тип `integer`;
  - `c` — тип `char`;
  - `f` — тип `float`;
  - `b` — тип `bit`.
- Ключевые поля начинаются с префикса `id`, далее идет имя таблицы, для связи с которой предназначен ключ. Таким образом, связь всегда видна.
- Главный ключ, как и другие ключевые поля, начинается с префикса `id`, затем идет имя таблицы, для которой ключ является главным.

Итак, начнем создавать макет базы данных. Основную таблицу назовем `tbClient`. Для начала ограничимся в ней информацией о названии и адресе клиента. Но клиент — это не только фирма, это еще и люди. Любой продавец должен иметь данные о контактных лицах клиента, с которыми можно при необходимости связаться. Такими лицами могут быть директор, заместители по направлениям, бухгалтеры и начальники отделов по направлениям. Например, если вы работаете в компании по продаже компьютеров, вам необходимо знать не только информацию о директоре, но и о начальниках информационных отделов, которые могут влиять на закупки и которым необходимо высылать рекламные буклеты.

Так как у одного клиента может быть несколько контактных лиц, информацию о них необходимо вынести в отдельную таблицу и организовать связь «один (клиент) ко многим (контактным лицам)».

У одного контактного лица может быть несколько телефонов. Среди них могут быть рабочий, стационарный и сотовый телефоны. Значит, снова необходимо выделять телефоны в отдельную таблицу и организовывать связь «один (контактное лицо) ко многим (номерам телефонов)».

Мы уже отметили, что телефон может быть стационарным и сотовым. Необходимо указывать это. Чтобы не приходилось каждый раз указывать тип телефона, их необходимо вынести в отдельный справочник.

Чтобы увидеть схему базы данных, с которой мы будем работать, я подготовил диаграмму в MS SQL Server (рис. 3.1). Напротив каждого поля дано его описание.

С левой стороны показы связи между таблицами. В нашей базе для начала будет еще три таблицы, которых нет на схеме:

- `tbJobPosition` — названия должностей, которые будут использоваться в таблице контактных лиц. Таблица состоит из двух полей — ключевого поля и поля названия должности.
- `tbTown` — названия городов. Чтобы пользователь не набирал названия городов вручную и таким образом исключить ошибки, лучше всего завести справочник, который будет состоять из двух полей — ключевого поля и поля названия города.
- `tbStreet` — названия улиц. Здесь понадобится три поля: ключевое поле, поле связи с таблицей городов и поле названия улицы. Улицы должны быть связаны с городом, потому что их будет очень много, а отображать улицы, которых нет в определенном городе, бессмысленно.

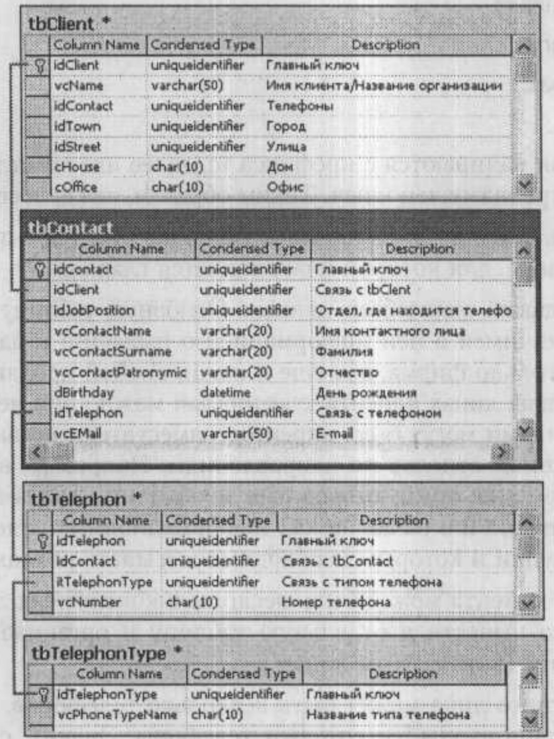


Рис. 3.1. Схема базы данных

Внимательно изучите схему. Вы должны разобраться во всех связях и понять, зачем они нужны. Базу данных необходимо знать назубок.

Обратите внимание, что в качестве ключей я использую тип `uniqueidentifier`. Это глобально уникальный идентификатор (Global Unique Identifier, GUID). Для главного ключа свойство `IsRowGuid` устанавливается в `true`. Это не обязательно,

достаточно просто прописать в свойство `Default Value` значение `(newid())`. Здесь вызывается функция `newid`, которая генерирует идентификатор. Таким образом, если идентификатор не создан, то его генерирует SQL Server.

ПРИМЕЧАНИЕ

Вам не обязательно создавать базу данных вручную. На компакт-диске в каталоге Sources\ch03\Intro есть резервная копия базы данных в файле db_backup.bak. Восстановите ее и используйте.

Многие программисты любят использовать в качестве ключей автоматически увеличивающиеся поля. Я не рекомендую этого делать для баз данных, с которыми будут работать несколько пользователей одновременно.

Почему я использую идентификаторы GUID? Они решают две основные проблемы, с которыми вы столкнетесь при работе с другим типами ключевых полей.

Первая проблема связана с *идентификаторами строк*. Когда мы добавляем в таблицу строку, идентификатора у нее нет. Это значит, что пока не сохранены изменения, идентификатор не появится и к строке нельзя будет добавлять подчиненные строки. Без идентификатора нельзя создать связь между строками в разных таблицах. Если использовать автоматически увеличиваемые поля, то проблема решается добавлением строки, моментальным сохранением данных и переходом в режим редактирования. А что если пользователь откажется от своих действий и попытается их отменить? Как выполнить откат изменений, которые уже сохранены в базе? Самое просто решение — удалить строку, но простота приводит к проблемам, потому что приходится хранить два кода, для добавления и для редактирования строки. А GUID мы легко можем сгенерировать самостоятельно и ввести в ключевое поле без помощи сервера.

Вторая проблема относится к *репликации данных*. При наличии двух серверов баз данных необходимо как-то организовывать взаимодействие данных. Что делать, если на обоих серверах пользователи создадут строки с одинаковым значением ключевого поля? Какую из записей записать в общую базу, в которой две базы данных объединяются в одну? Определить это невозможно, а изменять ключевое поле нельзя, потому что нарушатся связи. В случае с автоматически увеличиваемыми полями приходится резервировать на каждом сервере диапазоны для создания идентификаторов новых строк. С GUID такой проблемы быть не может, потому что вероятность создания двух строк с одинаковым ключевым полем практически нулевая. По крайней мере, Microsoft обещает, что появление двух одинаковых идентификаторов GUID возможно только теоретически.

Идентификаторы GUID несколько неудобны при отладке, потому что их сложно запоминать. Но преимущества покрывают все недостатки. Корпорация Microsoft рекомендует в качестве ключевого поля именно GUID, и в данном случае я согласен с этой рекомендацией. В данной главе вы убедитесь в этом.

Прежде чем переходить к решению задачи, посмотрим, как в таблице `tbClient` хранится адрес. Он хранится в виде отдельных составляющих: город, улица, дом, офис. Почему я предпочитаю такой вариант хранения? Однажды компания,

где я работал, должна была сдавать отчетность в государственные органы, требующие именно такого представления адреса, а в нашей базе данных весь адрес хранился в одном поле. Мы потратили месяц на разбиение поля адреса. Автоматически удалось обработать только 70 % строк, а все остальные строки имели какие-то специфичные особенности, которые не поддавались автоматической обработке. Привести в порядок 50 тысяч строк в короткие сроки нереально, поэтому сдать вовремя данные не удалось.

В то же время собрать адрес в одно целое — дело пяти минут. Поэтому всегда разбивайте адрес на отдельные поля, даже если на первый взгляд в этом нет необходимости. Никогда не знаешь, что потребуется в будущем.

Для создания приложения мы будем использовать двухуровневую технологию клиент–сервер. По возможности, решения некоторых задач будем выносить на сервер в виде хранимых процедур, но основную логику реализуем на клиенте.

При создании приложения постараемся использовать минимум форм. Хотя большинство программистов «лепит» формы по делу и без, при множестве форм управлять большим проектом становится просто невозможно. Это тот случай, когда визуальные возможности среды разработки плохо влияют на качество кода. Каждая лишняя форма:

- ухудшает читабельность проекта;
- увеличивает объем приложения;
- увеличивает время загрузки, причем попытки уменьшить количество автоматически создаваемых при старте приложения форм приводят лишь к минимальной экономии;
- порождает большое количество одинаковых фрагментов кода, повторяющихся в разных формах.

В нашем приложении будет минимум форм и максимум действий. Вы увидите, что универсальность позволит решить многие проблемы и сделать разработку проще, быстрее и надежнее.

Реализация справочников

Начнем со справочников. На главной форме создайте строку меню, в которой должно быть меню Справочники с пунктами Типы телефонов, Должности, Города, Улицы.

Создавать отдельное окно для работы с каждым из справочников — бессмысленная трата времени и сил. А что если у нас будет 20 справочников? Тогда количество окон окажется таким, что работать с ними станет невозможно. Есть способ лучше — написать универсальное решение, которое позволит работать с любым справочником. Лучше один раз попотеть, зато потом можно будет добавлять новые справочники одной строкой кода. В нашем случае понадобится всего два окна:

- окно для просмотра справочника, поиска данных и вызова основных команд;
- универсальное окно для редактирования строк.

Итак, начнем с первого окна. Создадим форму, которая будет называться `TDBDirectoryTemplateForm`. На форме нам понадобится: инструментальная панель с кнопками для вызова основных команд, сетка для просмотра данных и строка состояния (рис. 3.2). Поиск строк в окне справочника реализуем с помощью фильтров компонента `TADOTable`.

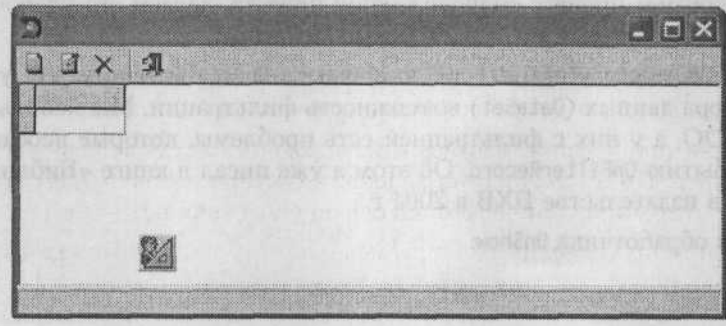


Рис. 3.2. Окно для работы со справочником

Как будет происходить создание и отображение окна? Поскольку код вызова каждого справочника будет одинаковым, давайте сразу же вынесем его в отдельный метод. Для этого в главной форме объявим метод `BuildGridDatabaseDirectoryItem` и реализуем его так, как показано в листинге 3.1.

Листинг 3.1. Метод отображения окна справочника

```
procedure TMainClientsForm.BuildGridDatabaseDirectoryItem(sCaption:String;
  ds: TDataSource);
var
  DirectoryForm:TDBDirectoryTemplateForm;
begin
  // Создаем форму
  DirectoryForm:=TDBDirectoryTemplateForm.Create(Owner);
  DirectoryForm.TableDBGrid.DataSource:=ds;

  DirectoryForm.Caption:=sCaption;
  DirectoryForm.ShowModal;

  DirectoryForm.Free;
end;
```

Сначала мы динамически создаем окно справочника. Затем для сетки `DBGrid` окна работы со справочником устанавливаем нужный источник данных `DataSource`. Так как источник зависит от вызываемого справочника, соответствующее значение будем передавать методу через параметр. Затем изменяем заголовок окна, чтобы пользователю было видно, с какими данными он работает. Заголовок также передается через параметр.

Дальнейшие действия должны быть понятны — это просто отображение окна и уничтожение после закрытия. Все остальное выполняет форма по работе со справочником.

Для вызова окна редактирования данных определенной таблицы справочника достаточно выполнить такую строку:

```
BuildGridDatabaseDirectoryItem('Типы телефонов',
  DataModule1.dsPhoneType);
```

Таким вот хитрым способом можно реализовать любое количество справочников. Для этого достаточно одной формы и одного метода для ее отображения. Так мы упрощаем процесс сопровождения проекта, делаем его удобочитаемым, уменьшаем объем кода и повышаем быстродействие.

На форме TDBDirectoryTemplateForm по событию OnShow включим для указанного в сетке набора данных (DataSet) возможность фильтрации. Мы используем компоненты ADO, а у них с фильтрацией есть проблемы, которые необходимо решать по событию OnFilterRecord. Об этом я уже писал в книге «Библия Delphi», вышедшей в издательстве БХВ в 2004 г.

Реализация обработчика OnShow:

```
var
 ds:TDataSet;
begin
 ds:= TableDBGrid.DataSource.DataSet;
 Filter:='';
 // Устанавливаем обработчик для OnFilterRecord
 TADOTable(ds).OnFilterRecord:=tbdOfficeFilterRecord;
 // Включить фильтрацию
 TADOTable(ds).Filtered:=true;
end;
```

Самое сложное здесь в том, что во время выполнения мы не знаем, с каким именно справочником работаем и какую таблицу надо использовать. Единственное средство связи с таблицей — это сетка. При создании окна сетка связывается с определенной таблицей через свойство DataSource. Чтобы получить доступ к таблице, будем использовать эту связь:

```
TableDBGrid.DataSource.DataSet
```

Строковая переменная Filter объявляется в разделе private формы, и в этой переменной будет храниться текущий фильтр поиска.

Фильтрация

Фильтрация — достаточно интересный момент, который надо разобрать более подробно, потому что путем фильтрации пользователи могут искать данные. Давайте сделаем так, чтобы можно было фильтровать данные по любому столбцу. Для этого создадим обработчик события OnKeyPress для сетки DBGrid и напишем в нем код, представленный в листинге 3.2.

Листинг 3.2. Фильтрация данных

```
procedure TDBDirectoryTemplateForm.TableDBGridKeyPress(Sender: TObject;
  var Key: Char);
var
 ds:TDataSet;
begin
// Для удобства сохраним в переменной таблицу, которая установлена в сетке
```

```
  ds:=TableDBGrid.DataSource.DataSet;

 // Если нажата кнопка Backspace, то нужно стереть последний символ
 // из строки фильтра
  if Key=#8 then
   begin
    Filter:=Copy(Filter, 1, Length(Filter)-1);
    if Filter='' then
     TADOTable(ds).Filtered:=false
    else
     TADOTable(ds).Filter :=
          TableDBGrid.SelectedField.FieldName+'>'''+ Filter+'''';
   end;

 // Если нажата русская клавиша или цифра, то добавляем ее к строке фильтра
  if ((Key>='а') and (Key<='я')) or
    ((Key>='А') and (Key<='Я')) or
    ((Key>='0') and (Key<='9')) then
   begin
    TADOTable(ds).Filtered:=true;
    Filter := Filter+Key;
    TADOTable(ds).Filter:=
          TableDBGrid.SelectedField.FieldName+'>'''+ Filter+'''';
   end;

 // Отображаем в строке состояния текущую строку фильтра
  if Filter<>'' then
   StatusBar1.SimpleText:=
     'Фильтр: '+TableDBGrid.SelectedField.DisplayName+'='+Filter
  else
   StatusBar1.SimpleText:='Фильтр: пуст';
 end;
```

Здесь мы добавляем в свойство `Filter` таблицы символ нажатой клавиши. Фильтр обрабатывает только цифры и буквы русской раскладки. Если вам потребуется обрабатывать другие символы и буквы другой раскладки, то соответствующий фрагмент необходимо добавить в код.

Если нажата клавиша Backspace, то последний символ из фильтра удаляется. Текущий фильтр для удобства отображается в строке состояния.

В качестве колонки, по которой необходимо фильтровать, используется текущая выделенная колонка. Получается, что для поиска строки со значением в определенной колонке пользователю необходимо установить курсор в эту колонку и просто начать набирать слово. По мере ввода число строк в сетке будет сокращаться — отображаться будут только те строки, которые соответствуют фильтру.

Мы уже знаем, что фильтры у компонентов ADO обрабатываются не совсем правильно, поэтому используем обработчик события `OnFilterRecord`, который будет выполняться по этому событию (листинг 3.3).

Фильтр из переменной `Filter` сравнивается со значением текущей строки. Если найдено соответствие, то строка разрешается (`Accept:=true`), иначе — запрещается (`Accept:=false`).

Листинг 3.3. Обработчик события OnFilterRecord

```
procedure TDBDirectoryTemplateForm.tbdOfficeFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
var
 Str:String;
begin
 if TableDBGrid.SelectedField=nil then exit;
 if TableDBGrid.SelectedField.Value=NULL then exit;

 Str:=TableDBGrid.SelectedField.Value;
 if AnsiUpperCase(copy(Str, 1, Length(Filter)))=AnsiUpperCase(Filter) then
  Accept:=true
 else
  Accept:=false;
end;
```

Вот таким хитрым способом мы реализовали универсальную фильтрацию. Чтобы сетка отображалась корректно, необходимо правильно настроить таблицы. Например, в модуле данных у вас должна быть создана таблица tbTown (компонент типа TADOTable), настроенная на таблицу городов базы данных. Дважды щелкните на этом компоненте, чтобы открыть редактор полей. Добавьте в редактор все поля (щелкните правой кнопкой мыши и выберите в контекстном меню команду Add all fields). Далее необходимо выполнить следующие настройки:

- спрятать ключевые поля, установив в свойстве Visible значение false (пользователям не нужно видеть ключи);
- указать на русском языке имена текстовых и числовых полей в свойстве DisplayLabel (эти имена будут отображаться в заголовках колонок);
- спрятать второстепенные ключевые поля и вместо них добавить поисковые поля (Lookup field) — это поможет нам при создании универсального окна редактирования;
- задать ширину колонок в свойстве DisplayWidth так, чтобы все поля были видны в окне для работы со справочником.

Редактирование данных

Теперь переходим к окну редактирования. Здесь все еще более запутано, потому что это окно должно обеспечивать редактирование строк с любым количеством полей и с любыми типами данных.

Назовем окно редактирования EditDBDirectoryForm. В нем будут динамически создаваться элементы управления для доступа к полям текущей строки. Высота окна должна быть достаточной для того, чтобы разместить в нем все видимые поля (поля, которые можно редактировать, без ключевых полей) в столбик. В нашем примере в справочниках пока что максимальное количество полей равно 2 (в таблице улиц).

Окно можно украсить, чтобы с ним приятно было работать. Кроме того, поместим на него две кнопки OK и Отмена.

Я стараюсь предложить универсальный пример, который поможет вам в будущем решать другие задачи, поэтому предусмотрел возможность редактирования

не только текстовых полей, но и полей для хранения цвета. Возможно, впоследствии они нам пригодятся.

Чтобы не писать лишний код, немного упростим задачу редактирования цвета. Ограничимся условием, что в таблице может быть только одно поле для хранения цвета. Просто поместим на форму одну кнопку с тремя точками в качестве заголовка (по щелчку на ней будет выводиться окно выбора цвета) и компонент ColorDialog, предназначенный для отображения самого окна (рис. 3.3).

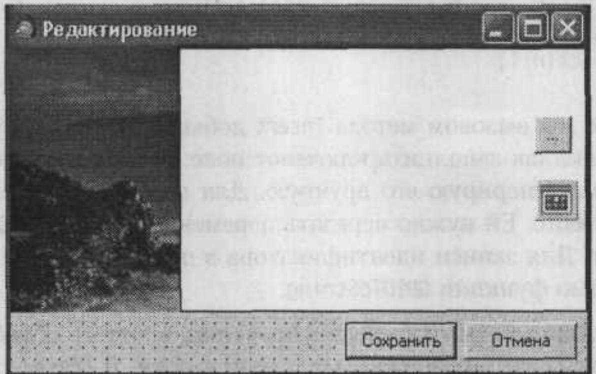


Рис. 3.3. Окно редактирования строк таблицы

Сразу же создадим обработчик события OnClick для кнопки:

```
if ColorDialog1.Execute then
  if ColorB<>nil then
   TDBEdit(ColorB).Text:=ColorToString(ColorDialog1.Color);
```

Что такое ColorB? Это переменная типа TDBEdit, в которой хранится указатель на поле для ввода кода цвета. Так как мы ограничились только одним полем, то можно воспользоваться такой переменной. Если в вашем справочнике потребуется больше полей, придется создавать их динамически, как мы это будем делать с другими полями.

Все, с окном редактирования закончили, можно возвращаться к окну просмотра данных DBDirectoryTemplateForm. Там у вас уже должны быть кнопки на панели инструментов для вызова операций добавления, редактирования и удаления записей из таблицы справочника.

Самый простой код выполняется при удалении записи:

```
procedure TDBDirectoryTemplateForm.DelButtonClick(Sender: TObject);
begin
 if Application.MessageBox(PChar(Удалить - "'
     +TableDBGrid.Columns[0].Field.AsString+'"'),
     'Внимание!!!',
     MB_ICONINFORMATION+MB_OKCANCEL)<>ID_OK then exit;
 TableDBGrid.DataSource.DataSet.Delete;
end;
```

Здесь мы всего лишь выводим запрос на подтверждение удаления и, если ответ положителен, вызываем метод Delete текущего справочника.

Теперь посмотрим на процесс добавления строки. По щелчку на кнопке добавления строки выполняется следующий код:

```
procedure TDBDirectoryTemplateForm.AddButtonClick(
  Sender: TObject);
var
 gdNew:TGUID;
begin
 TableDBGrid.DataSource.DataSet.Insert;
 CreateGUID(gdNew);
 TableDBGrid.DataSource.DataSet.Fields[0].AsString:=
      GUIDToString(gdNew);
 EditButtonClick(nil);
end;
```

В первой строке мы вызовом метода Insert добавляем в таблицу новую запись. Чтобы вы поняли, как заполнить ключевое поле еще на этапе создания строки, в этом примере я генерирую его вручную. Для генерации GUID используется процедура CreateGUID. Ей нужно передать переменную типа TGUID, куда будет записан результат. Для записи идентификатора в поле я привожу его тип к строковому с помощью функции GUIDToString.

Добавив строку, вызываем окно редактирования, а точнее обработчик события щелчка на кнопке редактирования выделенной строки. Я всегда так делаю, чтобы не приходилось дважды писать код открытия окна. В данном случае эта проблема стоит наиболее остро, потому что из-за необходимости динамически создавать компоненты в окне код отображения окна получился достаточно большой (листинг 3.4).

Листинг 3.4. Обработчик события OnClick для кнопки редактирования строки

```
procedure TDBDirectoryTemplateForm.EditButtonClick(Sender: TObject);
var
 i:Integer;
 l:TLabel;
 edit:TDBEdit;
 lookup:TDBLookupCombobox;
 EditDBDirectoryForm:TEditDBDirectoryForm;
begin
 // Создание окна редактирования
 EditDBDirectoryForm:=TEditDBDirectoryForm.Create(Owner);

 //Прячем кнопку выбора цвета и обнуляем указатель на поле ввода цвета
 EditDBDirectoryForm.SelectColorButton.Visible:=false;
 EditDBDirectoryForm.ColorB:=nil;

 // В цикле обрабатываем все колонки справочника
 for i:=0 to TableDBGrid.Columns.Count-1 do
  begin
   // Создаем подпись TLabel с названием поля
   l:=TLabel.Create(EditDBDirectoryForm.Owner);
   // Задаем параметры подписи TLabel с названием поля
   l.Parent:=EditDBDirectoryForm.MainPanel;
   l.Left:=8;
   l.Top:=(i+1)*8+i*21+2;
```

```
l.Transparent:=true;
// Заголовок для подписи берется из свойства DisplayLabel поля
l.Caption:=TableDBGrid.Columns[i].Field.DisplayLabel;

// Если это поисковое поле, для его редактирования
// создаем компонент TDBLookupCombobox
if TableDBGrid.Columns[i].Field.FieldKind=fkLookup then
 begin
  lookup:=TDBLookupCombobox.Create(EditDBDirectoryForm.Owner);
  lookup.Parent:=EditDBDirectoryForm.MainPanel;
  lookup.Left:=120;
  lookup.Width:=145;
  lookup.Top:=(i+1)*8+i*21;
  lookup.DataSource:=TableDBGrid.DataSource;
  lookup.DataField:=TableDBGrid.Columns[i].Field.FieldName;
  Continue;
 end;

// Для остальных компонентов создается TDBEdit
edit:= TDBEdit.Create(EditDBDirectoryForm.Owner);
edit.Parent:=EditDBDirectoryForm.MainPanel;
edit.Left:=120;
edit.Width:=145;
edit.Top:=(i+1)*8+i*21;
edit.CharCase:=ecUpperCase;
edit.DataSource:=TableDBGrid.DataSource;
edit.DataField:=TableDBGrid.Columns[i].Field.FieldName;

// Для поля, хранящего цвет, заголовок поля должен быть равен
// строке "Цвет". В этом случае делаем видимой кнопку отображения
// окна выбора цвета и размещаем ее напротив поля ввода
if l.Caption='Цвет' then
 begin
  edit.Width:=118;
  EditDBDirectoryForm.SelectColorButton.Top:=(i+1)*8+i*19;
  EditDBDirectoryForm.SelectColorButton.Visible:=true;
  EditDBDirectoryForm.ColorB:=edit;
 end
end;
// Отображаем окно редактирования
TableDBGrid.DataSource.DataSet.Edit;
EditDBDirectoryForm.ShowModal;

// Если выполнен щелчок на кнопке OK,
// сохраняем данные, иначе откатываем изменения
if EditDBDirectoryForm.ModalResult=mrOK then
 begin
  TableDBGrid.DataSource.DataSet.Post;
 end
else
 begin
  TableDBGrid.DataSource.DataSet.Cancel;
 end;
EditDBDirectoryForm.Free;
end;
```

В данном листинге мы динамически размещаем на форме, предназначенной для окна редактирования строк из справочника, пары компонентов TLabel и TDBEdit для каждого поля. Если найдено поисковое поле (в этом случае свойство FieldKind равно fkLookup), вместо компонента TDBEdit создается компонент TDBLookupComboBox. Подробные комментарии помогут вам разобраться с этим кодом.

Я рекомендую вам реализовать возможность динамического создания кнопки для выбора цвета. Для экономии места я немного упростил задачу, но если вы разберетесь с вопросами динамического создания кнопок и выбора цвета, то сможете создать и более сложные решения.

После формирования окна мы отображаем его на экране. Если пользователь щелкает на кнопке OK, то данные сохраняются, иначе вызывается метод Cancel набора данных, чтобы откатить изменения.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\1-References.

Работа с клиентами

Если справочники мы реализовали в модальных окнах и тем самым упростили себе жизнь, то для реализации основной функциональности будем использовать дочерние окна. Дочерние окна намного удобнее, но применение многодокументного интерфейса (Multiple Document Interface, MDI) ведет к определенным проблемам.

Поскольку наше приложение будет многодокументным, измените у главной формы свойство FormStyle на fsMDIForm. У дочерних окон это свойство должно устанавливаться в fsMDIChild.

Начнем с создания шаблонного окна. Постараемся сделать его максимально функциональным, чтобы потом использовать для работы с разными таблицами и запросами. На главной форме шаблонного окна пока ограничимся только сеткой и панелью с кнопками Добавить, Редактировать и Удалить (рис. 3.4).

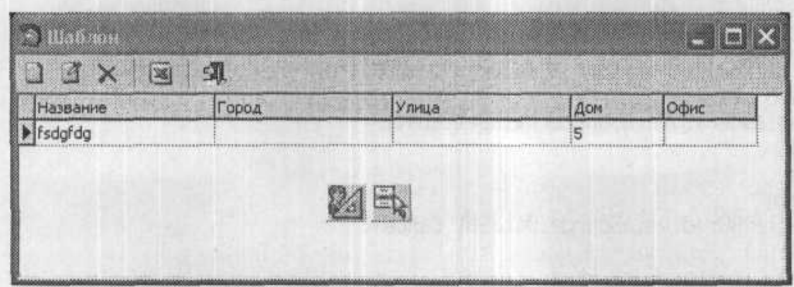


Рис. 3.4. Форма шаблонного клиентского окна

В качестве сетки я буду использовать компонент CyDDBGrid. Этот компонент я создал специально для своих бизнес-приложений. Главное его преимущество — возможность единственной командой выгружать данные сетки в Excel, что требуется достаточно часто. Кроме того, он позволяет выводить логические поля в виде флажков (CheckBox), отображать изображения прямо в сетке и имеет встроенный редактор Memo-полей. Исходный код компонента можно найти на компакт-диске в каталоге components/grid.

Шаблонное окно не будет знать, с какой таблицей мы работаем. Единственный способ получить доступ к таблице — связь через сетку:

```
dbgMainGrid.DataSource.DataSet
```

В данном случае dbgMainGrid — это имя сетки.

По щелчку на кнопке **Добавить** будет выполняться только метод Insert для таблицы, которая установлена в сетке:

```
procedure TChildTemplateForm.acAddExecute(Sender: TObject);
begin
 dbgMainGrid.DataSource.DataSet.Insert;
end;
```

Щелчок на кнопке **Редактировать** обеспечит переход в режим редактирования:

```
procedure TChildTemplateForm.acEditExecute(Sender: TObject);
begin
 dbgMainGrid.DataSource.DataSet.Edit;
end;
```

По умолчанию окна MDI создаются сразу при старте программы, и закрыть их нельзя. Чтобы решить проблему закрытия, необходимо убрать форму из числа автоматически создаваемых и в обработчики события OnClose окон добавить строку:

```
Action:=caFree;
```

Для создания окна на главной форме добавим пункт меню **Файл ▸ Клиенты**. По событию OnClick для этого пункта меню будет выполняться код:

```
var
 ClientForm:TChildTemplateForm;
begin
 ClientForm:=TChildTemplateForm.Create(Self);
 ClientForm.dbgMainGrid.DataSource:=DataModule1.dsClients;
 ClientForm.Caption:='Клиенты';
end;
```

В первой строке мы динамически создаем шаблонное окно. Вторая строка устанавливает в качестве значения свойства DataSource компонент DataModule1.dsClients. Это компонент TADOQuery, в котором выполняется следующий запрос:

```
SELECT *
FROM tbClient cl, tbTown t, tbStreet s
WHERE cl.idTown*=t.idTown AND cl.idStreet*=s.idStreet
```

Этот запрос реализует простейшую связь трех таблиц. Обратите внимание, что первой идет таблица tbClient. Первая таблица редактируется без проблем, поля остальных таблиц изменять нельзя. Об этом мы еще поговорим, потому что

удалить строку из запроса, в котором есть связанные таблицы, нельзя, так как неизвестно, из какой таблицы удалять строку.

Теперь реализуем возможность отображения окна редактирования данных о клиенте. В отличие от наших справочников в реальной программе в таблице клиентов будет намного больше полей. Да и окно будет открываться пользователями достаточно часто, поэтому его желательно сделать удобным, отвести для него отдельную форму и расположить все поля в удобном для работы виде (рис. 3.5).

Рис. 3.5. Форма редактирования данных клиентов

Для полей Название, Дом и Офис используется компонент DBEdit. Для поля Город лучше подойдет компонент DBLookupComboBox, потому что городов чаще всего не так уж и много, и их можно выбирать в списке. А вот улиц очень много, и поиск в списке может быть затруднительным, поэтому для поля Улица пока установим простой компонент ComboBox и чуть позже напишем для него очень интересный код.

Как и где отображать это окно? Встраивать в шаблонное окно нельзя, тогда создание шаблона окажется бессмысленным. Необходимо вынести этот код куда-нибудь в другое место. Лучшим вариантом является перенос кода в модуль данных. Создадим обработчик события AfterEdit для компонента qClients (это запрос на выборку данных клиентов) и в нем отобразим созданное нами окно редактирования данных клиента. Событие AfterEdit генерируется, когда таблица переходит в режим редактирования, то есть вызывается метод Edit, что у нас происходит по щелчку на кнопке Редактировать.

При вставке строк тоже желательно отображать окно редактирования, в котором намного удобнее вводить данные. Вставьте тот же фрагмент в код, выполняемый по событию AfterInsert, которое генерируется после вызова метода Insert.

Суперпоисковые поля

В разделе «Редактирование данных» мы создали окно для редактирования данных о клиенте. Для ввода названий улиц у нас установлен простой компонент TComboBox, который не имеет связи с базой данных, а значит, вводимые данные

сохраняться в базе не будут. Всю запись будем заполнять вручную, но очень удобным способом.

Так как улиц в каждом городе много, то выбирать в списке справочника нужную строку неудобно. Проблемы возникают и в том случае, если вводимой улицы нет в справочнике. Пользователю придется лезть в справочник, добавлять нужную запись и снова возвращаться к редактированию данных о клиенте. Все это время пользователь будет вспоминать вас нелестными словами. В результате некоторые программисты просто отказываются от справочников, но это не выход. Я покажу вам очень удобный и гибкий вариант использования справочника.

Надо предоставить пользователю возможность вводить любое название улицы. По мере ввода в раскрывающемся списке будут появляться возможные варианты, а если введенного названия в базе не окажется, программа должна автоматически добавлять это название в справочник. Возможности компонента TLookupComboBox слишком скромны, поэтому его использовать нельзя. Будем писать свою реализацию компонента, используя в качестве основы простой компонент TComboBox, причем сделаем нашу реализацию максимально универсальной (вдруг вы захотите вводить названия банков, которых в России скоро будет столько же, сколько улиц).

Для реализации описанных возможностей нам понадобится три обработчика событий компонента TComboBox:

- OnEnter — сохранение текущего размера текста, введенного в поле компонента ComboBox;
- OnChange — при изменении текста в поле нужно найти варианты завершения ввода и предложить их пользователю;
- OnExit — сохранение в таблице введенных данных и, если необходимо, добавление в справочник нового названия.

Самое простое — это вход в компонент. Здесь будет выполняться всего одна строка:

```
cbLength:=Length(TComboBox(Sender).Text);
```

Переменная cbLength должна быть объявлена в разделе private вашей формы как целое число (Integer). Почему я так сложно (через параметр Sender) определяю размер? Чтобы сделать реализацию максимально универсальной. Если у вас на форме несколько компонентов ComboBox, в которых нужно будет реализовать описываемый здесь механизм поиска, то для всех можно установить один обработчик события OnEnter. Компонент, генерирующий это событие, передается в обработчик через параметр Sender. Так как этот параметр имеет тип TObject, мы явно указываем, что это компонент TComboBox, и определяем длину свойства Text.

Таким образом, в переменной cbLength всегда будет находиться размер активного в данный момент раскрывающегося списка.

При изменении данных должен выполняться вызов процедуры FindComboItems:

```
FindComboItems(cbStreet, 'tbStreet', 'vcStreetName',
   ' AND idTown='''+lcTown.KeyValue+'''', cbLength);
```

Что это за процедура? Мы сами должны будем ее реализовать. Для хранения реализации процедуры я создал модуль UtilsUnit.pas. Если в каком-либо проекте потребуется суперпоисковое поле, я просто добавлю его к проекту и после

минимальной доработки модуль будет готов к работе. Код процедуры представлен в листинге 3.5.

Листинг 3.5. Поиск возможных вариантов

```
procedure FindComboItems(cb: TComboBox; TableName,
  NameColumn, AddSQL: String; var cbLength:Integer);
begin
  // Если размер текста не изменился, то выход
  if cbLength=Length(cb.Text) then
   begin
    exit;
   end;

  // Если изменилось более одного символа, то выход
  if Length(cb.Text)-cbLength>1 then
   begin
    cbLength:=Length(cb.Text);
    exit;
   end;

  // Запоминаем текущий размер
  cbLength:=Length(cb.Text);

  // Если поле пустое, то ничего не ищем,
  // чтобы не выводить содержимое всего справочника.
  // Если справочник слишком большой, то можно
  // увеличить число до 2, тогда новые варианты
  // будут предлагаться пользователю
  // только после ввода двух символов

  if Length(cb.Text)<1 then
   exit;

  // Поиск возможных вариантов
  DataModule1.qrCommonSQL.Active:=false;
  DataModule1.qrCommonSQL.SQL.Text:='SELECT TOP 20'+nameColumn+
     ' FROM '+TableName+' WHERE '+nameColumn+
     ' LIKE ''%'+cb.Text+'%'''+AddSQL;
  DataModule1.qrCommonSQL.Active:=true;

  // Обновляем содержимое раскрывающегося списка,// заполняя его найденными
    значениями
  cb.Items.BeginUpdate;
  cb.Items.Clear;
  DataModule1.qrCommonSQL.First;
  while DataModule1.qrCommonSQL.Eof<>true do
   begin
    cb.Items.Add(DataModule1.qrCommonSQL.Fields[0].AsString);
    DataModule1.qrCommonSQL.Next;
   end;
  cb.Items.EndUpdate;

  // Выделяем
  cb.SelStart:=Length(cb.Text);
  cb.SelLength:=0;
end;
```

Для начала рассмотрим параметры, которые передаются в процедуру. Это поможет вам лучше понять, что в ней происходит.

- cb — указатель на компонент ComboBox, в котором пользователь вводит данные.
- TableName — имя таблицы справочника, в котором необходимо искать возможные значения. Это имя будет использоваться в запросе на выборку данных.
- NameColumn — имя поля в таблице, в котором находятся искомые названия. В нашем случае пользователь вводит названия улиц, а они в таблице tbStreet перечислены в поле vcStreetName.
- AddSQL — дополнительные SQL-ограничения. В нашем случае таблица tbStreet связана с таблицей tbTown (улицы привязаны к определенным городам), и мы должны искать только те улицы, которые находятся в выбранном городе. Дополнительный запрос как раз и налагает это ограничение.
- cbLength — размер текста в компоненте TComboBox во время предыдущего выполнения этой процедуры или, если выполнения не было, во время входа в компонент. Этот размер необходим для проверки на изменение. Например, если текст не изменился, символы удалены или слишком мало данных для поиска, то выборка данных производиться не будет в целях экономии процессорного времени.

Подробные комментарии в листинге 3.5 помогут вам разобраться с процедурой FindComboItems. Я ограничусь только несколькими замечаниями.

Что означает запись DataModule1.qrCommonSQL? Компонент qrCommonSQL — это компонент типа TADOQuery, используемый для выполнения простых запросов, возвращающих данные, которые не нужно долго хранить. Зачем для получения таких данных каждый раз создавать отдельный компонент TADOQuery? Для этого я завел единственный компонент qrCommonSQL и использую его по мере необходимости.

С помощью компонента qrCommonSQL выполняется поиск строк, соответствующих введенным пользователем словам. Обратите внимание, что в запросе на выборку данных я ограничиваю максимальное количество возвращаемых строк числом 20, чтобы количество элементов в раскрывающемся списке было не слишком большим. В большом списке выбирать трудно. Это значение можно увеличить, но не советую устанавливать его более чем 50. Найденные названия улиц добавляются в раскрывающийся список.

Теперь самое сложное — сохранение изменений. Это будем делать по событию OnExit списка. В нашем случае будем вызывать процедуру SaveComboItems следующим образом:

```
SaveComboItems(cbStreet, 'tbStreet', 'vcStreetName',
     'idStreet', ' AND idTown='''+lcTown.KeyValue+'''',
     'Такой улицы', DataModule1.tbStreet, 3,
     TField(DataModule1.qClients.FieldByName('idStreet')),
       DataModule1.tbStreetidTown, lcTown.KeyValue);
```

Сама процедура показана в листинге 3.6, и ее нужно реализовать в модуле UtilsUnit.

Листинг 3.6. Функция сохранения выбранного значения

```
procedure SaveComboItems(cb: TComboBox; TableName,
  NameColumn, KeyColumn, AddSQL, MessageText: String; ds: TDataSet;
  index: Integer; sf, sfDictionary: TField; idParent: String);
var
 gdNew:TGUID;
begin
  cb.DroppedDown:=false;

  if cb.Items.IndexOf(cb.Text)=-1 then
   begin
    // Проверка на существование указанного названия
    DataModule1.qrCommonSQL.Active:=false;
    DataModule1.qrCommonSQL.SQL.Text:='SELECT '+KeyColumn+
       ' FROM '+TableName+' WHERE '+NameColumn+' = '''+cb.Text+''''+AddSQL;
    DataModule1.qrCommonSQL.Active:=true;

    // Название не существует
    if DataModule1.qrCommonSQL.RecordCount=0 then
     begin
      if MessageBox(0, PChar(MessageText+' нет в справочнике. Добавить?'),
            'Внимание!!!', MB_ICONINFORMATION+MB_OKCANCEL)<>ID_OK then
       begin
        cb.Text:='';
        exit;
       end;

      // Добавление названия
      ds.Insert;
      CreateGUID(gdNew);
      ds.Fields[0].AsString:=GUIDToString(gdNew);
      ds.Fields[Index].AsString:=cb.Text;
      if sfDictionary<>nil then
        sfDictionary.AsString:=idParent;
      ds.Post;
     end
    else
     // Название существует, используем
     gdNew:=StringToGUID(DataModule1.qrCommonSQL.Fields[0].AsString);
   end
  else
   begin
    // Название существует, используем
    DataModule1.qrCommonSQL.Active:=false;
    DataModule1.qrCommonSQL.SQL.Text:='SELECT '+KeyColumn+
       ' FROM '+TableName+' WHERE '+NameColumn+' = '''+cb.Text+''''+AddSQL;
    DataModule1.qrCommonSQL.Active:=true;
    gdNew:=StringToGUID(DataModule1.qrCommonSQL.Fields[0].AsString);
   end;

  // Запись результата
  sf.AsString := GUIDToString(gdNew);
 end;
```

Рассмотрим параметры, которые передаются процедуре, чтобы вам проще было разобраться.

- cb — компонент ComboBox, в котором находится сохраняемое название.
- TableName — таблица справочника (в нашем случае это таблица tbStreet).
- NameColumn — колонка с названиями в справочнике (в нашем случае с названиями улиц).
- KeyColumn — ключевая колонка в справочнике.
- AddSQL — для связанных справочников добавочный SQL-код поиска (в нашем случае улицы привязаны к городу, поэтому добавляем этот код).
- MessageText — текст сообщения MessageBox. Это сообщение появляется в случае, если введенное название улицы не найдено и нужно запросить у пользователя разрешение на автоматическое добавление его в справочник.
- ds — набор данных (TDataSet) справочника (TTable или TQuery), в который нужно добавить новую запись. В нашем случае в набор данных будет добавлено новое введенное пользователем название улицы.
- index — индекс (номер) колонки с текстом. В нашем случае это поле vcStreetName.
- sf — колонка в основной таблице, которую надо изменить. В нашем случае это поле idStreet в таблице tbClient. Именно это поле связано со справочником.
- sfDictionary — поле в справочнике, связанное с другой таблицей. В нашем случае в справочнике есть поле idTown, которое связано с таблицей городов. При добавлении новой строки в справочник в это поле нужно записать идентификатор города, которому принадлежит добавляемая улица. Если таблица не связана, то в этом параметре необходимо передать значение nil.
- idParent — ключевое значение для связи с главной таблицей, которое нужно записать в поле, передаваемом в параметре sfDictionary.

Параметров очень много, но, как я уже отмечал, нам необходим максимально универсальный код, который можно будет использовать в любых проектах.

В начале процедуры мы закрываем раскрывающийся список (cb.DroppedDown:=false). Если он останется открытым, это будет выглядеть не слишком эстетично.

Затем проверяем, есть ли уже в списке введенное пользователем слово:

```
if cb.Items.IndexOf(cb.Text)=-1 then
```

Если введенного слова в списке нет, то на всякий случай проверяем базу данных. Для этого по указанному пользователем названию улицы ищем соответствующую строку. Если результат отрицателен, предлагаем пользователю добавить строку в справочник.

При добавлении строки мы вручную генерируем GUID для главного ключа. Это не обязательно, если для ключевого поля у вас по умолчанию выполняется генерация, но я генерирую GUID «вручную», потому что в будущих проектах у меня могут быть строки с отсутствующим идентификатором. Трудно предсказать, что может понадобиться в будущем.

В поле названий новой строки записываем введенное пользователем название улицы и, если есть, идентификатор города. Если бы улицы у нас не были привязаны к городу, можно было бы оставить идентификатор города нулевым — процедура работала бы точно так же.

Если введенный пользователем текст есть в списке, значит, он есть и в базе данных, остается только найти его идентификатор и использовать. Для этого выполняется запрос на поиск ключа для введенного названия улицы, и этот ключ сохраняется в переменной gdNew.

И наконец, содержимое переменной gdNew записываем в соответствующее поле таблицы клиентов.

Запустите программу, добавьте названия нескольких городов и во время редактирования в справочнике записей о клиентах попробуйте добавить названия нескольких улиц. Все должно работать нормально. После этого откройте справочник улиц и убедитесь, что все связи улица–город организованы правильно (рис. 3.6).

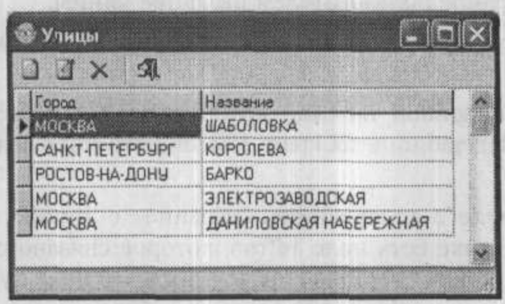


Рис. 3.6. Окно справочника улиц

Но это еще не все. При добавлении новой строки название улицы в окне редактирования отсутствует. Когда вы его добавляете, то все отображается верно, если заново считать данные, то есть снова выполнить запрос. Обновление сетки таблицы клиентов лучше всего делать по событию AfterPost (событие генерируется после сохранения изменений, сделанных в строке таблицы, то есть после выполнения метода Post):

```
DataSet.Active:=false;
DataSet.Active:=true;
```

Здесь мы просто закрываем запрос и открываем его заново. Таким образом, изменения, которые мы внесли в базу данных, корректно считываются. Но если открыть окно редактирования, поле списка TComboBox окажется пустым. Поэтому перед открытием окна необходимо вручную его заполнить. В нашем случае это достаточно просто. Ведь запрос, который выполняется для заполнений сетки значениями, уже содержит название улицы, поэтому отображение окна может быть реализовано следующим образом:

```
procedure TDataModule1.qClientsAfterEdit(
    DataSet: TDataSet);
begin
```

```
  EditClientForm.cbStreet.Text:=
     qClientsvcStreetName.AsString;
  EditClientForm.ShowModal;
 end;
```

В данном случае qClientsvcStreetName — это указатель на поле vcStreetName. К полю можно обратиться и по его имени:

```
EditClientForm.cbStreet.Text:=
    qClients.FieldByName('vcStreetName').AsString;
```

Но это — самый простой вариант. А если бы запрос возвращал не название улицы, а только ее идентификатор? В этом случае пришлось бы искать имя улицы по идентификатору. Таким образом, универсальная процедура отображения окна редактирования должна выглядеть так, как показано в листинге 3.7.

Листинг 3.7. Процедура отображения окна клиентов с поиском имени улицы

```
procedure TDataModule1.qClientsAfterEdit(DataSet: TDataSet);
begin
 EditClientForm.cbStreet.Text:='';
  // Если улица задана (ключевое поле не пустое), то
  if DataModule1.qClients.FieldByName('idStreet').AsString<>'' then
   begin
    // Ищем по ключу название улицы
    DataModule1.qrCommonSQL.Active:=false;
    DataModule1.qrCommonSQL.SQL.Text:=
          'SELECT vcStreetName FROM tbStreet WHERE idStreet = '''+
           DataModule1.qClients.FieldByName('idStreet').AsString+'''';
    DataModule1.qrCommonSQL.Active:=true;

    // Если поиск дал результат, копируем его в раскрывающийся список
    if DataModule1.qrCommonSQL.RecordCount>0 then
     EditClientForm.cbStreet.Text:=
          DataModule1.qrCommonSQL.Fields[0].AsString;
   end;

  EditClientForm.ShowModal;
 end;
```

Вот теперь пример можно считать законченным, потому что с ним уже можно работать. Осталось только для удобства наделить программу еще несколькими дополнительными возможностями, и ее можно будет назвать полноценной.

> ПРИМЕЧАНИЕ
>
> Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\2-References.

Управление запросами

При программировании для баз данных очень важно эффективно хранить SQL-запросы, поскольку от этого зависит, насколько удобной получится программа и насколько простым будет ее код.

Если есть возможность, то запросы желательно выносить в хранимые процедуры и функции сервера. Если запрос статичный или в нем меняются один-два

параметра, то я всегда выношу такой запрос на сторону сервера. Большинство серверов выполняют хранимые процедуры намного быстрее, потому что могут кэшировать план их выполнения и не производить разбор запроса перед каждым запуском.

Если запрос очень часто изменяется, то программисты стараются его генерировать во время выполнения программы. Это ошибка, которая приводит к усложнению сопровождения. Если запрос должен кардинально меняться (как это будет в нашем случае при добавлении в программу возможностей поиска и сортировки), то его придется хранить на стороне клиента. Однако ни в коем случае не храните его в самой программе. Однажды я видел исходный код большой складской программы, где все SQL-запросы были включены в код. Таких запросов было более 100, и после изменения любого из них приходилось перекомпилировать программу.

Текст запросов лучше всего хранить в отдельном файле. Я для этого завожу каталог `SQL` и создаю в нем текстовые файлы, которые подгружаются в программу динамически. Такой способ хранения имеет очевидные преимущества.

- Запрос можно изменить прямо во время выполнения программы. Я всегда подгружаю текстовый файл во время создания клиентского окна, в котором будет использоваться запрос. Если я замечаю ошибку или просто хочу оптимизировать работу, мне достаточно изменить файл и заново создать клиентское окно. При этом саму программу перезапускать не надо.
- Запросы можно обновлять без остановки клиентских компьютеров. Пользователи могут продолжать работать в момент обновления.
- Изменяя текст запроса, можно наращивать его возможности без перекомпиляции программы.

Однако у такого способа хранения SQL-запросов есть и один недостаток. Обновляя программы, я иногда забываю обновить текстовые файлы запросов. Из-за этого новые версии могут не работать или пользователи просто не получат доступа к новым возможностям.

Как хранить запрос, который часто изменяется, чтобы не генерировать его во время выполнения программы? В текстовый файл необходимо записать шаблон, в котором будут три секции:

- `SELECT` с перечислением выбираемых полей;
- `FROM` с перечислением таблиц, используемых в запросе для выборки данных;
- `WHERE` с описанием основных связей.

Каждая секция должна занимать ровно одну строку. Тогда если придется изменить одну из секций, при подгрузке программы достаточно будет изменить соответствующую строку, не затрагивая другие.

Чаще всего изменяется секция `WHERE`, в которую могут добавляться дополнительные условия отбора данных. Чтобы не корректировать третью строку, я всегда в ней храню только объявление связей. Дополнительные критерии отбора переносятся в 5–7 строки. Например, вот как может выглядеть запрос для выборки клиентов в нашей программе:

```
1 SELECT *
2 FROM tbClient cl, tbTown t, tbStreet s
3 WHERE cl.idTown*=t.idTown AND cl.idStreet*=s.idStreet
4
5 Поиск
6 Поиск
7 Поиск
8
9
10 Сортировка
```

Первые три строки остаются неизменными, а вот остальные могут изменяться в зависимости от требований клиента. Строки 4, 8 и 9 резервируются для особых случаев (если потребуется что-то оригинальное).

В следующих двух разделах мы на практике рассмотрим, как можно реализовать сортировку и поиск. А сейчас вам необходимо вынести текст запроса из компонента TADOQuery и сохранить его в файле SQL/clients.sql. Первые три строки должны быть такими, как в предыдущем примере, а остальные оставим пустыми. Всего в файле должно быть 10 строк, иначе у вас возникнут ошибки при работе с остальными примерами книги.

Загрузку запроса выполним перед созданием окна просмотра списка клиентов, как показано в листинге 3.8.

Листинг 3.8. Создание окна просмотра клиентов

```
procedure TMainClientsForm.acClientsExecute(Sender: TObject);
var
  ClientForm:TChildTemplateForm;
Begin
  // Создание окна
  ClientForm:=TChildTemplateForm.Create(Self);

  // Связь сетки с компонентом запроса TADOQuery
  ClientForm.dbgMainGrid.DataSource:=DataModule1.dsClients;

  // Загрузка и выполнение запроса
  DataModule1.qClients.Active:=false;
  DataModule1.qClients.SQL.LoadFromFile(
        ExtractFilePath(Application.ExeName)+'SQL\clients.sql');
  DataModule1.qClients.Active:=true;

  // Изменение заголовка
  ClientForm.Caption:='Клиенты';
 end;
```

Сортировка

Теперь перейдем к организации сортировки. Это не будет простым добавлением секции Orders в SQL-запрос для определенного поля. Мы намного усложним задачу. Пользователь должен иметь возможность выполнять сортировку по любому полю, а точнее — по любым полям, в том числе сразу по нескольким, а также по убыванию и по возрастанию. В строке состояния (добавьте ее на форму)

будем отображать текущие поля, по которым отсортирована таблица, и направление сортировки.

Сортировку будем выполнять по щелчку на кнопке панели инструментов окна или на заголовке сетки. В случае сетки по событию `OnTitleClick` должен выполняться код из листинга 3.9.

Листинг 3.9. Обработчик события OnTitleClick для компонента сетки DBGrid

```
procedure TChildTemplateForm.dbgMainGridTitleClick(Column: TColumn);
var
 Index: Integer;
begin
  // Запоминаем текущую выделенную колонку
  Index:=dbgMainGrid.SelectedIndex;
  // Функция генерации строки сортировки
  UniversalSortGrid(Column,
     StatusBarl.Panels[0], SortPointer, SortString,
     TADOQuery(dbgMainGrid.DataSource.DataSet).SQL);

  // Перечитываем данные
  TADOQuery(dbgMainGrid.DataSource.DataSet).Active:=false;
  TADOQuery(dbgMainGrid.DataSource.DataSet).Active:=true;

  // Восстанавливаем выделенную колонку
  dbgMainGrid.SelectedIndex:=Index;
end;
```

Так как запрос будет выполняться заново, необходимо запомнить выделенную колонку, чтобы потом восстановить выделение. Чтобы не было проблем с восстановлением, я рекомендую в сетке располагать поля в том же порядке, что и в таблице. Все ключевые и невидимые поля, кроме поля основного ключа, необходимо перенести в самый конец. Для таблицы клиентов все поля, имена которых начинаются с префикса `id` (невидимые ключевые поля), я перенес в самый конец (рис. 3.7). Напоминаю, чтобы отрыть показанный на рисунке редактор, нужно дважды щелкнуть на компоненте `TADOQuery` или `TTable`.

Рис. 3.7. Редактор полей в TADOQuery

Самое основное происходит в процедуре `UniversalSortGrid`. Ее необходимо реализовать в модуле `UtilsUnit`, чтобы использовать в любых последующих формах и проектах. Код этой процедуры показан в листинге 3.10.

Листинг 3.10. Процедура универсальной сортировки

```
procedure UniversalSortGrid(Column: TColumn; sb:TStatusPanel;
  var SortPointer:Boolean;
  var Sort:String; SQL: TStrings);
var
 Additional, StatusAdd:String;
 State: TKeyboardState;
begin
 // Это вторая сортировка?
 GetKeyboardState(State);
 if ((State[vk_Shift] and 128) <> 0) and (Sort<>'') then
  begin
   Additional:=Sort+', ';
   StatusAdd:=sb.Text+', ';
  end
 else
  begin
   Additional:='';
   StatusAdd:='Отсортировано: ';
  end;

 // Это изменение направления сортировки?
 if Pos(Column.FieldName, Sort)>0 then
  begin
   SortPointer:=not SortPointer;
   if Additional<>'' then
    Delete(Additional, Pos(Column.FieldName, Additional),
        Length(Column.FieldName)+7);
    Delete(StatusAdd, Pos(Column.Field.DisplayLabel, StatusAdd),
        Length(Column.Field.DisplayLabel)+5);
  end
 else
  SortPointer:=true;

 // Сортируем
 if SortPointer then
  begin
   Sort:= Additional+Column.FieldName+' ASC';
   sb.Text:=StatusAdd + Column.Field.DisplayLabel+' ^ ';
  end
 else
  begin
   Sort:= Additional+Column.FieldName+' DESC';
   sb.Text:=StatusAdd + Column.Field.DisplayLabel+' V ';
  end;
 SQL[10]:='ORDER BY '+Sort;
end;
```

Но прежде чем рассматривать эту процедуру, давайте добавим возможность сортировки по щелчку на кнопке. Пока что у нас создан только обработчик события щелчка на заголовке. По щелчку на кнопке сортировки должен выполняться следующий код:

```
dbgMainGridTitleClick(
    dbgMainGrid.Columns[dbgMainGrid.SelectedIndex]);
```

Здесь мы вызываем метод `dbgMainGridTitleClick`. На самом деле, это обработчик события `OnTitleClick`, который мы уже написали. Чтобы не писать код заново,

я просто передаю управление уже имеющемуся методу. В качестве параметра необходимо передать выделенную колонку, что и обеспечивает конструкция, которую мы передаем в параметре:

```
dbgMainGrid.Columns[dbgMainGrid.SelectedIndex]
```

Разберемся сначала с параметрами, которые передаются в процедуру.

- Column — выделенная колонка. По этому параметру в процедуре будет определяться имя поля, по которому должна выполняться сортировка, и заголовок поля, который выводится в строке состояния (компоненте StatusBar).
- sb — указатель на панель в строке состояния, в которой нужно отобразить подсказку о сортируемых полях и направлении сортировки.
- Sort — строковая (string) переменная, в которой хранится код сортировки. Эту переменную нужно объявить в разделе private, а по событию OnShow присваивать ей значение по умолчанию:

  ```
  SortString := 'ORDER BY idClient';
  ```

- SortPointer — логическая переменная, которая определяет направление сортировки (по возрастанию или по убыванию). Эту переменную нужно объявить в разделе private как имеющую тип boolean, а по событию OnShow присваивать ей значение по умолчанию.

Давайте посмотрим, что происходит в самой процедуре. Сначала вызывается WinAPI-функция GetKeyboardState, которая возвращает состояние клавиш. Нас интересует клавиша Shift. Если при щелчке мышью на заголовке она была нажата, это означает, что пользователь хочет добавить колонку в число сортируемых. То есть механизм выделения столбцов будет таким же, как и при выделении файлов (чтобы выделить несколько файлов, мы щелкаем на их именах, удерживая клавишу Shift).

Тут необходимо заметить, что выделять колонки с помощью клавиши Shift можно только щелчками на их заголовках. Использовать совместно с клавишей Shift кнопку сортировки на панели инструментов для выделения колонок не удастся.

Итак, если нажата клавиша Shift, то в переменной Additional сохраняются текущие поля сортировки, и в этом случае выделенное поле добавляется в эту переменную. В переменной StatusAdd сохраняется текущий текст строки состояния, а также имена отсортированных полей.

Далее мы проверяем, есть ли в переменной Sort имя выделенной колонки. Если есть, то ее добавлять не надо. Нужно просто поменять направление сортировки. Для этого изменяем значение переменной SortPointer на противоположное и удаляем из строки, содержащей поля для сортировки, уже существующее имя, потому что оно будет добавлено, но с другим направлением сортировки.

Теперь все готово к окончательному формированию переменной Sort, которая содержит перечисление сортируемых колонок, и выводу имен колонок в строке состояния.

Самая последняя строка кода копирует в десятую строку запроса сформированный список полей. В разделе «Управление запросами» мы уже отмечали, что в запросе нужно создать десять строк и последняя будет использоваться как раз для хранения секции ORDER BY.

Запустите программу, откройте окно просмотра списка клиентов и выделите какое-нибудь поле. Щелкните на заголовке сетки или нажмите клавишу F2. Содержимое выбранного поля должно быть отсортировано, а в строке состояния должен появиться текст:

```
Отсортировано: Название поля ^
```

Если щелкнуть еще раз, то направление сортировки изменится и в строке состояния можно будет увидеть:

```
Отсортировано: Название поля v
```

Знак ^ или v после названия поля показывает направление сортировки. Попробуйте выполнить сортировку нескольких полей и убедитесь, что все работает корректно. Но для этого придется создать не менее десять строк с различными значениями во всех полях, иначе трудно будет проверить правильность сортировки.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\3-Sort.

Эффективный поиск

Теперь займемся реализацией механизма поиска. Очень часто, открыв окно для работы с таблицей, пользователям приходится искать данные в определенных колонках. В этом разделе нам предстоит написать код, который позволит наделить нашу программу универсальными и гибкими возможностями поиска. Для этого будет создана новая форма, которая сможет искать данные в любом компоненте `DBGrid` любого окна. Сколько бы у вас ни было форм, чтобы наделить их возможностями поиска, потребуется только несколько строк кода.

На рисунке 3.8 показана форма для будущего окна поиска. Давайте назовем эту форму `FindInGridForm`. Вверху окна выводится информация о том, по какому полю будет происходить поиск. В раскрывающемся списке **Условие** выбирается условие поиска; доступные варианты: =, >, <, >=, <=, != и **Диапазон**. В поле **Искать** пользователь будет вводить искомый текст. При выборе в списке **Условие** пункта **Диапазон** в списке **До** потребуется выбрать конечное значение поиска.

Рис. 3.8. Форма для окна поиска

По умолчанию поиск является мягким. Это означает, что к искомому значению в начале и в конце будет добавляться символ процента (%). При установке флажка **Название целиком** поиск станет жестким, то есть мы будем искать точное вхождение указанного слова.

В списках **Искать** и **До** скрыты компоненты для выбора даты DateTimePicker, применяемые для поиска в полях типа datetime. В качестве самих списков я использовал компонент TMaskEdit, а не TEdit, чтобы можно было устанавливать маску для ввода дат.

Теперь посмотрим, как выполнять поиск с помощью такой формы. В окне для работы с клиентами добавим кнопку поиска и напишем следующий код, который должен выполняться при щелчке на этой кнопке:

```
FindInGridForm.FindInGrid:=dbgMainGrid;
FindInGridForm.ShowModal;
StatusBar1.Panels[1].Text:='Записей: '+IntToStr(
   TADOQuery(dbgMainGrid.DataSource.DataSet).RecordCount);
```

В первой строке записываем в свойство FindInGrid формы поиска указатель на сетку. Так как поиск универсален, то есть не зависит от таблицы или запроса, программа должна знать, в какой сетке находятся данные, и через эту сетку получать доступ к компоненту TADOQuery. Кстати, мы еще не создали это свойство. В разделе public формы поиска объявите его следующим образом:

```
FindInGrid : TDBGrid;
```

Далее отображаем форму поиска. После этого в строке состояния должно выводиться количество найденных строк.

Переходим к рассмотрению самой формы поиска. По событию OnShow должен выполняться код из листинга 3.11.

Листинг 3.11. Обработчик события OnShow формы поиска

```
procedure TFindInGridForm.FormShow(Sender: TObject);
begin
 FindLabel.Caption:='Поиск по: '+FindInGrid.SelectedField.DisplayLabel;

 // Вызываем обработчик события OnChange
 // для раскрывающегося списка "Условие"
 cbFindConditionChange(nil);

 // Устанавливаем фокус в поле ввода искомого значения
 FindEdit.SetFocus;
 FindEdit.SelectAll;

 // Если тип поля, по которому происходит поиск,
 // относится к дате, устанавливаем
 // маску, облегчающую ввод даты
 if (FindInGrid.SelectedField.DataType=ftDate) or
    (FindInGrid.SelectedField.DataType=ftDateTime) then
  begin
   FindEdit.EditMask:='99/99/9999';
   FindForEdit.EditMask:='99/99/9999';
  end
 else
```

```
  begin
   FindEdit.EditMask:='';
   FindForEdit.EditMask:='';
  end;
 end;
```

Давайте посмотрим, что происходит в этом коде. Сначала в заголовок окна мы записываем имя выделенного поля. По нему выполняется поиск, поэтому лучше, чтобы пользователь всегда его видел.

Затем мы вызываем обработчик события OnChange для раскрывающегося списка **Условие**. Здесь выполняется проверка; если выделен элемент **Диапазон**, то делается доступным поле ввода **До**, иначе оно блокируется. Вторая проверка делает доступными компоненты DateTimePicker, если искомое поле является датой. Весь этот код можно увидеть в листинге 3.12. Со всем остальным, я надеюсь, вы разберетесь по комментариям. Как всегда, я постарался сделать их максимально подробными.

Листинг 3.12. Реализация поиска

```
procedure TFindInGridForm.cbFindConditionChange(Sender: TObject);
begin
 // Если выбран диапазон, то сделать его доступным,
 // иначе заблокировать, чтобы не мозолил глаза
 if cbFindCondition.ItemIndex=RangeCondition then
  begin
   FindForEdit.Enabled:=true;
   cbAllString.Enabled:=false;
   FindForEdit.Color:=clWhite;
  end
 else
  begin
   FindForEdit.Enabled:=false;
   cbAllString.Enabled:=true;
   FindForEdit.Color:=clBtnFace;
  end;

 cbAllString.Enabled:=false;
 DateTimePicker1.Enabled:=false;
 DateTimePicker2.Enabled:=false;

 case FindInGrid.SelectedField.DataType of
  // Отображать элемент "Поле целиком", только если тип данных строковый
  ftWideString,ftString:
    cbAllString.Enabled:=true;

 // Отображать компоненты DateTimePicker, только если поиск
 // происходит по полю типа Date или DateTime
 ftDate, ftDateTime:
   begin
    DateTimePicker1.Enabled:=true;
    DateTimePicker2.Enabled:=true;
   end;
 end;
end;
```

А вот теперь переходим к самому интересному — формированию самого условия поиска и выполнению запроса. Как вы помните, под поиск с помощью нашего универсального окна зарезервирована седьмая строка SQL-запроса. Именно в нее добавляются все условия выборки данных.

Код обработчика события щелчка на кнопке Найти можно увидеть в листинге 3.13. Я постарался сделать этот код максимально простым, чтобы вам легче было его понять и, если необходимо, модернизировать под свои нужды.

Листинг 3.13. Поиск данных по выбранным пользователем условиям

```
procedure TFindInGridForm.FindActionExecute(Sender: TObject);
var
 FieldName, Condition, Text1, Text2:String;
 ft:TFieldType;
 SelectedIndex:Integer;
begin
 FieldName:=FindInGrid.SelectedField.FieldName;
 SelectedIndex:=FindInGrid.SelectedField.Index;
 TRY
 ft:=FindInGrid.SelectedField.DataType;

 TADOQuery(FindInGrid.DataSource.DataSet).Active:=false;

 Condition:=' '+cbFindCondition.Items[cbFindCondition.ItemIndex]+' ';

 if (ft=ftWideString) or
    (ft=ftString) then
  begin
   ///////////////////////////////////////////////////
   // Поиск строки
   if cbFindCondition.ItemIndex=0 then
    Condition:=' LIKE ';

   if cbFindCondition.ItemIndex=RangeCondition then
    begin
     TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:=
           ' AND '+FieldName+' >= '''+FindEdit.Text+''' AND '+
           FieldName+' <= '''+FindForEdit.Text+''''
    end
   else
    begin
     if (cbAllString.Checked) or (cbFindCondition.ItemIndex<>0) then
      TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:=
           ' AND '+FieldName+Condition+''''+FindEdit.Text+''''
     else
      TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:=
           ' AND '+FieldName+Condition+'''%'+FindEdit.Text+'%''';
    end;
  end
 else
  begin
   ///////////////////////////////////////////////////
   // Поиск остальных типов данных
   if (ft=ftDate) or
      (ft=ftDateTime) then
    begin
```

```
try
  Text1:=FormatDateTime('mm/dd/yyyy', StrToDate(FindEdit.Text));
  if cbFindCondition.ItemIndex=RangeCondition then
   Text2:=FormatDateTime('mm/dd/yyyy', StrToDate(FindForEdit.Text));
 except
  Application.MessageBox('Неправильно указана дата',
         'Ошибка', MB_ICONINFORMATION+MB_OK);
  TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:='';
  TADOQuery(FindInGrid.DataSource.DataSet).Active:=true;
  FindInGrid.SelectedIndex:=SelectedIndex-1;
  exit;
 end;
 end
else
 begin
  Text1:=FindEdit.Text;
  Text2:=FindForEdit.Text;
 end;

 if (ft=ftInteger) or
    (ft=ftSmallint) then
  begin
   if cbFindCondition.ItemIndex=RangeCondition then
    TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:=
       ' AND '+FieldName+'>='+Text1+' AND '+FieldName+' <= '+Text2
   else
   TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:=
         ' AND '+FieldName+Condition+Text1;
  end
 else
  begin
   if cbFindCondition.ItemIndex=RangeCondition then
    TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:=
      ' AND '+FieldName+'>='''+Text1+''' AND '+FieldName+' <= '''+
      Text2+''''
   else
    TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:=
      ' AND '+FieldName+Condition+''''+Text1+'''';
  end
 end;

 TADOQuery(FindInGrid.DataSource.DataSet).Active:=true;
 FindInGrid.SelectedIndex:=SelectedIndex-1;
 if TADOQuery(FindInGrid.DataSource.DataSet).RecordCount>0 then
  begin
   ModalResult:=mrOK;
  end
 else
  Application.MessageBox('Не найдено!', 'Внимание!!!', MB_OK);
 EXCEPT
  Application.MessageBox('Неправильно указаны данные!',
      'Ошибка', MB_ICONINFORMATION+MB_OK);
  TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:='';
  TADOQuery(FindInGrid.DataSource.DataSet).Active:=true;
  FindInGrid.SelectedIndex:=SelectedIndex-1;
  exit;
 end;
end;
```

Весь код разбит на фрагменты в зав мости от типа поля, в котором происходит поиск. Внимательно изучите ко гобы понять, как происходит поиск.

На самом деле в этой процедуре ничего сложного нет. Зная SQL и основы Delphi, ее можно написать за полчаса с перерывами на чашечку кофе. После изучения примера я советую заглянуть на диск и посмотреть полный исходный код. В него для удобства я добавил несколько фрагментов, которые могут пригодиться вам в будущем.

Запустив и протестировав пример, вы заметите, что поиск городов и улиц не выполняется. Даже если вы правильно введете названия улицы, все строки останутся на месте. Почему именно эти поля? Посмотрите на запрос, который используется в качестве основы и к которому мы добавляем критерии поиска:

```
SELECT *
FROM tbClient cl, tbTown t, tbStreet s
WHERE cl.idTown*=t.idTown AND cl.idStreet*=s.idStreet
```

Проблема кроется в секции `WHERE`. Связь между таблицами не жесткая (используется конструкция `*=`). Из-за этой звездочки в знаке операции в таблице слева можно выводить строки, даже если для них не найдено соответствие в таблице справа. Если звездочку указать справа (`=*`), все действует наоборот.

Когда мы задаем название определенной улицы и пытаемся выполнить поиск, из-за нежесткой связи данные о клиентах с других улиц все равно могут выводиться на экран. Чтобы все работало верно, необходимо заменить знак `*=` простым равенством:

```
WHERE cl.idTown=t.idTown AND cl.idStreet=s.idStreet
```

Однако в этом случае не будут отображаться данные о клиентах, у которых не заполнено поле улицы или города, потому что в этих записях нет связи с соответствующими справочниками. А так как у нас установлено жесткое равенство между таблицами, то, если где-то нет связи, такая строка не выводится.

В результате приходится выбирать одно из следующих решений.

- Установить жесткую связь. В этом случае поиск по именам из справочников станет возможным, но нельзя будет найти строки для клиентов, у которых поле улицы или города окажется незаполненным.
- Установить мягкую связь (левую или правую). В этом случае можно будет найти всех клиентов, но не удастся выполнить поиск по именам улиц и городов.
- Для поиска названия улицы сначала найти ее ключ в справочнике (например, для улицы `idStreet`) и в условии поиска потребовать искать этот идентификатор в таблице клиентов (добавить в запрос строку `tbClient.idStreet=`*идентификатор*). Но в этом случае сложно добиться универсальности, поскольку непонятно, как определить, какие имена полей в сетке взяты из справочника, а какие из основной таблицы.
- Убрать связь со справочником из запроса и сократить его до строки `SELECT * FROM tbClient`. Для определения названия улицы или города использовать поисковые поля. Перед началом поиска легко определить, что это поисковое поле, по его типу (это мы уже делали), и благодаря этому заменить значение названия в колонке ее ключом. Это я не стал реализовывать в своей

процедуре, а оставил вам в качестве домашнего задания. Попробуйте сами это сделать.

В программу можно добавить кнопку, по щелчку на которой поисковая строка будет обнуляться. Такую кнопку можно назвать Показать все. Для очистки строки поиска достаточно в седьмую строку запроса записать пустое значение (листинг 3.14).

Листинг 3.14. Очистка строки поиска

```
procedure TFindInGridForm.CearFindActionExecute(Sender: TObject);
var
 SelectedIndex:Integer;
begin
 SelectedIndex:=FindInGrid.SelectedField.Index;

 TADOQuery(FindInGrid.DataSource.DataSet).Active:=false;
 TADOQuery(FindInGrid.DataSource.DataSet).SQL[7]:='';
 TADOQuery(FindInGrid.DataSource.DataSet).Active:=true;

 FindInGrid.SelectedIndex:=SelectedIndex-1;
end;
```

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\4-Search.

Шаблон клиентского окна

Теперь наше шаблонное клиентское окно обладает всеми основными возможностями, которые могут понадобиться пользователю. В нем есть возможность добавления и редактирования записей, сортировки и поиска, экспорта в Excel (рис. 3.9). Чтобы наделить нашу программу новыми возможностями, достаточно создать новый запрос TADOQuery наподобие того, который мы написали для клиента, и отобразить шаблонное окно с этим запросом. Созданное нами окно останется полностью функциональным и будет корректно работать без внесения изменений в код.

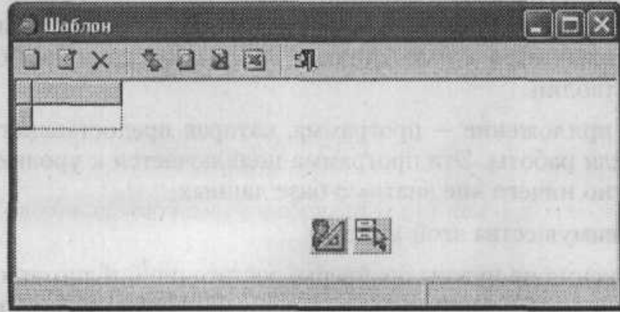


Рис. 3.9. Результат разработки шаблона клиентского окна

Единственное, что вы должны дописать, — код для динамического создания окна редактирования выделенной строки, как мы это делали в справочниках. В таблице клиентов я показал, как заставить шаблонное окно открывать нестандартные окна через обработчики событий AfterEdit и AfterInsert и при этом не терять универсальности. Динамическое создание окон можно вынести в эти же обработчики событий ваших запросов. Если вы сделаете этот код универсальным, как это было в справочниках, то один обработчик сможет обрабатывать события нескольких компонентов TADOQuery.

Трехуровневые приложения

Трехуровневые приложения большинство авторов и программистов почему-то обходят стороной. А зря. В случае больших приложений трехуровневая структура может быть действительно полезной, поскольку предоставляет дополнительные возможности при развертывании. Если вы пишете приложение для предприятия, то вполне логично использовать в нем три уровня. Почему? Попробую в этом разделе показать преимущества этой технологии и убедить вас в необходимости ее применения.

Прежде чем продолжать чтение, я рекомендую вам заглянуть в документ dblevel.pdf, находящийся в каталоге Doc\Database прилагаемого компакт-диска. В этом документе я постарался описать достоинства и недостатки сетевой клиент-серверной трехуровневой модели.

Основы трехуровневых приложений

Так как литературы по трехуровневым приложениям для баз данных не слишком много, я постараюсь уделить этому вопросу больше внимания. Для начала посмотрим, из чего состоят три уровня приложения (рис. 3.10).

- База данных — любая база данных, например MS Access, MS SQL Server, Oracle и др.
- Уровень логики — на этом уровне реализуется логика работы приложения, чаще всего это сервер приложений. Умные люди придумали этому уровню красивое название — уровень бизнес-логики. Только этот уровень непосредственно подключается к базе данных. Здесь происходит выборка и изменение данных таблиц.
- Клиентское приложение — программа, которая предоставляет пользователю интерфейс для работы. Эта программа подключается к уровню логики и может абсолютно ничего «не знать» о базе данных.

Рассмотрим преимущества этой модели.

- Для подключения не нужны драйверы, достаточно библиотеки midas.dll, которую не нужно даже регистрировать в системе. Нужно только поместить этот файл в каталог, доступный программе.

- Для изменения логики работы программы очень часто достаточно обновить сервер, не трогая клиентские приложения, поскольку вся логика реализована на сервере.
- Клиентские приложения могут создаваться по разным технологиям. Например, одни клиенты могут быть Win32-приложениями, другие Linux-приложениями, третьи работать через окно браузера. Для этих трех приложений придется написать только разный визуальный интерфейс, а логика их работы будет общей для всех.

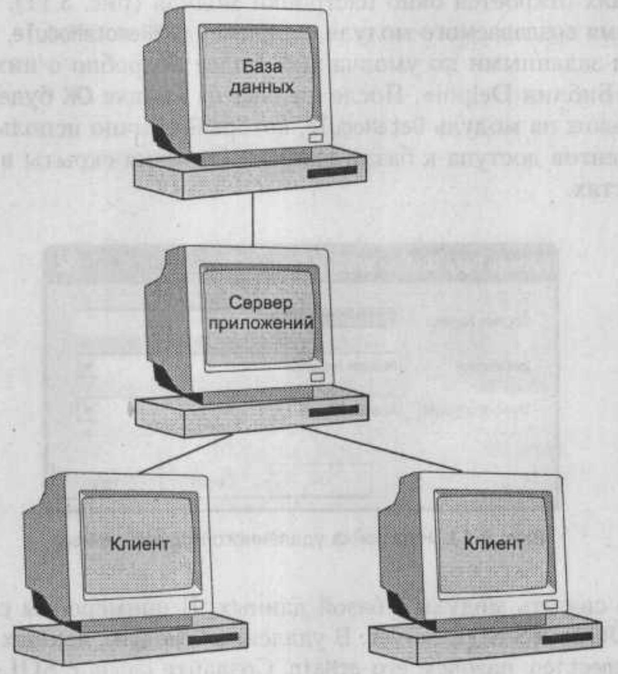


Рис. 3.10. Схема трехуровневой модели

Эти достоинства с лихвой компенсируют единственный замеченный мной за все время работы с базами данных существенный недостаток трехуровневых приложений — сложность разработки. Нет, создать сервер, реализующий логику, не так уж и сложно. Проблема в том, что приходится писать две отдельные программы, чему я долго противился, поскольку привык реализовывать «все в одном». Однако недавно по просьбе заказчика мне пришлось писать трехуровневое приложение, и в ходе его разработки я сумел оценить всю мощь подобной структуры.

Пример, который я описывал в уже неоднократно упоминавшейся книге «Библия Delphi», был слишком прост, в нем были реализованы только основы. В данном разделе мы познакомимся с созданием сервера бизнес-логики более глубоко. Давайте *создадим сервер бизнес-логики и клиента для тестирования примера*, чтобы оценить достоинства этой технологии на реальном примере.

Создание сервера

Создавать сервер мы начнем с написания простого приложения для платформы Win32. На главной форме пока ничего располагать не будем, а сразу же создадим удаленный модуль данных (remote data module). Для этого выберите в меню команду File ▸ New ▸ Other. В Delphi 7 значок Remote Data Module расположен на вкладке Multitier. В Delphi 2005 нужно в дереве разделов выбрать пункт Delphi Projects, раскрыть раздел ActiveX и в этом разделе вы найдете значок Remote Data Module.

В обоих случаях откроется окно настройки модуля (рис. 3.11). Введите в поле CoClass Name имя создаваемого модуля, например CRMRemoteModule. Остальные значения оставим заданными по умолчанию. Более подробно о них можно прочитать в книге «Библия Delphi». После щелчка на кнопке OK будет создан новый модуль. Он похож на модуль DataModule, который обычно используется для хранения компонентов доступа к базам данных. Отличия скрыты в исходном коде и в возможностях.

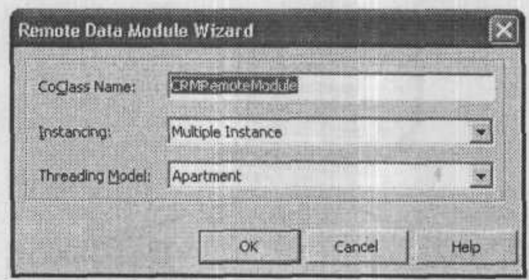


Рис. 3.11. Настройка удаленного модуля данных

Теперь нужно связать модуль с базой данных. В примере мы снова будем использовать ADO и MS SQL Server. В удаленный модуль данных поместим компонент TADOConnection, назовем его acMain. Создайте связь с SQL-сервером и базой данных, как описано в разделе «Постановка задачи».

Теперь добавим компонент TADOQuery с запросом на выборку клиентов, который мы также использовали в предыдущих примерах. В клиент-серверных приложениях для отображения данных мы добавляли компонент TDataSource. Приложения бизнес-логики не будут отображать данные на форме, поэтому этот компонент нам не нужен. Однако нам требуется сделать компонент TADOQuery доступным другим приложениям. Для этого используется компонент TDataSetProvider, который чем-то схож с TDataSource, но предназначен не для отображения таблицы в компонентах доступа к данным, а для публикации данных в клиентских приложениях.

Установим на форме компонент TDataSetProvider (вкладка DataAccess) и в его свойстве DataSet укажем компонент с запросом к таблице клиентов. На этом этапе наш запрос еще недоступен клиентам. Чтобы сделать его доступным, необходимо изменить свойство Exported на true, и тогда компонент DataSetProvider позволит клиентам использовать запрос.

Вроде бы простейший сервер готов, и можно переходить к созданию клиента для тестирования примера. Но мы же пишем универсальное решение, а что может быть более универсальным, чем возможность подключения к любой базе данных и возможность настройки этого подключения. В случае универсального решения закладывать имя сервера уже на этапе создания сервера нельзя — нужно предоставить пользователю средства для его редактирования без перекомпиляции программы. Как это сделать?

В модуле ADOConEd есть функция EditConnectionString, которой нужно передать компонент AdoConnection, тогда на экран будет выведено стандартное окно настройки подключения. Проблема в том, что модуль удаленных данных создается в памяти только в момент подключения клиента к серверу, а настраивать подключение надо заранее.

Для решения проблемы установим на главную форму еще один компонент ADOConnection. Поскольку главная форма уж точно создается при старте программы, компонент мы сможем редактировать. По событию OnCreate для формы должен выполняться код из листинга 3.15.

Листинг 3.15. Обработчик события OnCreate для формы

```
procedure TCRMServerForm.FormCreate(Sender: TObject);
var
 FIniFile: TRegIniFile;
begin
 // Создаем объект для работы с реестром
 FIniFile := TRegIniFile.Create('Software');
 FIniFile.OpenKey('Flenov',true);
 FIniFile.OpenKey('CMS Server',true);

 // Читаем строку подключения
 acMain.ConnectionString:=FIniFile.ReadString('Option', 'Connection', '');

 // Если строка пустая, то редактируем ее
 if acMain.ConnectionString='' then
  begin
   EditConnectionString(acMain);
  end;

 // В ConnectEdit типа TEdit отображаем строку подключения
 ConnectEdit.Text:=acMain.ConnectionString;

 // Пробуем подключиться
 try
  acMain.Connected:=true;
 except
  MessageBox(0, 'Подключение невозможно', 'Ошибка', MB_OK+MB_ICONERROR );
 end;
end;
```

Самостоятельно напишите код сохранения, который должен выполняться при закрытии формы, используя в качестве основы код чтения строки подключения к базе данных.

Чтобы пользователь в любой момент мог изменить строку подключения, на форму можно добавить кнопку. Код обработки щелчка на кнопке выглядит следующим образом:

```
acMain.Connected:=false;
EditConnectionString(acMain);
ConnectEdit.Text:=acMain.ConnectionString;
```

Перед редактированием строки подключения делаем соединение неактивным, иначе произойдет ошибка.

В результате у нас на главной форме программы должно быть поле ввода для отображения строки подключения, кнопка для изменения строки и компонент TADOConnection (рис. 3.12).

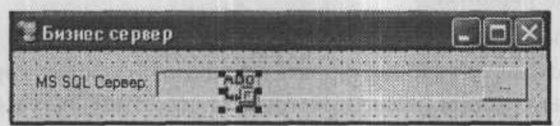


Рис. 3.12. Окно главной программы сервера бизнес-логики

А теперь самое интересное — как использовать строку подключения главной формы в удаленном модуле данных. Выделите сам модуль, тогда на вкладке Evens объектного инспектора можно будет увидеть событие OnCreate. Это событие генерируется при подключении клиента и создании модуля. В качестве значения этого свойства нужно указать такую же строку подключения, как и на главной форме. Мой обработчик события можно увидеть в листинге 3.16.

Листинг 3.16. Обработчик события OnCreate удаленного модуля данных

```
procedure TCRMRemoteModule.RemoteDataModuleCreate(Sender: TObject);
var
 i:Integer;
begin
 // Если подключение активно, то выходим. Менять уже ничего не надо
 if acMain.Connected=true then exit;

 // Изменяем строку подключения и соединяемся с сервером
 acMain.ConnectionString:=CRMServerForm.acMain.ConnectionString;
 acMain.Connected:=true;

 // Активизируем все таблицы
 for i:=0 to ComponentCount-1 do
  if (copy(Components[i].Name, 1, 2)='tb') and
     (Components[i].ClassName='TADOTable') then
    TADOTable(Components[i]).Active:=true;
end;
```

Обратите внимание, что если соединение уже активно, то происходит выход, поэтому компонент acMain (это компонент типа TADOConnection на форме модуля данных) при старте программы должен быть неактивным. Для этого в дизайнере форм отключите свойство Connected. Чтобы не забыть про этот нюанс, перед компиляцией я даже очищаю свойство ConnectionString. Оно необходимо, чтобы

с таблицами можно было работать в дизайнере (без него нельзя будет получать поля таблиц и запросов). Но перед компиляцией строка удаляется.

В Delphi 2005 я заметил одну особенность — если хотя бы одна таблица (или запрос) в дизайнере активна, то происходит следующее.

1. При подключении клиента активизируется и мое соединение acMain, причем еще до того, как выполняется обработчик события OnCreate и устанавливается правильная строка подключения.
2. Так как в компоненте acMain даже нет строки подключения, появляются сообщения об ошибках типа: «База данных не найдена».
3. Только после этого обрабатывается событие OnCreate, и подключение устанавливается.

Чтобы не столкнуться с этой проблемой, ни один компонент в дизайнере форм не должен быть активным — активация должна происходить во время работы. Я выполняю активацию по событию OnCreate.

Если запросы автоматически активируются при обращении к ним клиентов, то таблицы должны быть уже активированными. Для этого я запускаю цикл, в котором перебираются все компоненты типа TADOTable на форме, и если имя такого компонента начинается с префикса tb, то его свойство Active устанавливается в true.

В нашем примере пока нет таблиц и только один запрос. Но уже сейчас необходимо заложить возможность роста будущего проекта.

Теперь для работы сервера его нужно зарегистрировать в системе, иначе клиенты не смогут его найти и подключиться. Для этого просто запустите программу сервера, и регистрация произойдет автоматически. После этого сервер можно закрывать, потому что при подключении со стороны клиента операционная система сама его запустит.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\Levels\Server.

Создание клиента

Для создания клиента разработаем пустое приложение Delphi для Win32 и модуль данных DataModule. В модуле данных нам понадобится соединение с сервером бизнес-логики. Для этого воспользуемся компонентом TSocketConnection с вкладки DataSnap.

Почему именно этот компонент? Он проще в развертывании и управлении. На стороне клиента достаточно будет настроить протокол TCP/IP (сейчас уже трудно встретить компьютер без настроенного протокола) и запустить утилиту scktsrvr.exe, которую можно найти в каталоге bin каталога установки Delphi.

У самого компонента необходимо указать адрес компьютера, на котором зарегистрирован сервер бизнес-логики. Если вы все делаете на одном компьютере, то в свойстве Address укажите адрес 127.0.0.1. После этого раскройте список значений свойства ServerName, и если вы верно зарегистрировали сервер, то в списке будет его имя.

Подключение клиента к серверу также нельзя прописывать в исходном коде. А вдруг изменится IP-адрес компьютера или сервер бизнес-логики перенесут на другой компьютер? Что, снова перекомпилировать клиентскую программу? Нет, это не выход. Перед компиляцией клиента нужно очищать поле адреса, все компоненты делать неактивными, а подключение создавать на этапе загрузки программы. Давайте посмотрим, как это сделать.

Во всех своих программах я всегда пишу две процедуры:

- SaveProgParam — сохранение параметров программы. Эта процедура вызывается по событию OnClose главной формы. В ней сохраняются все необходимые параметры (в том числе адрес сервера бизнес-логики и расположение окон);
- LoadProgParam — загрузка сохраненных параметров.

Пример процедуры LoadProgParam для клиентского приложения показан в листинге 3.17. Код, который восстанавливает положение и состояние окна, я опустил, оставив только то, что касается адреса сервера бизнес-логики.

Листинг 3.17. Загрузка параметров программы

```
procedure TClientForm.LoadProgParam;
var
 FIniFile: TRegIniFile;
 sAddr:String;
 i:Integer;
 bChangeAddress:Boolean;
begin
 FIniFile := TRegIniFile.Create('Software');
 FIniFile.OpenKey('Flenov',true);
 FIniFile.OpenKey('Client',true);

 // Здесь можно реализовать чтение параметров окон,
 // включая положение, состояние и т. д.

 // Чтение параметров соединения с бизнес-логикой
 DataModule1.stConnection.Connected:=false;
 DataModule1.stConnection.Address:=
   FIniFile.ReadString('Option', 'ConnectionAddress', '');

 // Если программу запустили с ключом /s, то нужно сменить адрес сервера
 bChangeAddress:=false;
 for i := 1 to ParamCount do
  if LowerCase(ParamStr(i)) = '/s' then
   bChangeAddress:=true;

 // Если адрес не указан или нужно сменить адрес
 // сервера, запрашиваем у пользователя новый адрес
 if (DataModule1.stConnection.Address='') or (bChangeAddress=true) then
  begin
   InputQuery('Адрес сервера', 'Введите, пожалуйста, адрес сервера', sAddr);
   DataModule1.stConnection.Address:=sAddr;
  end;

 FIniFile.Free;
end;
```

В этой процедуре мы считываем строку подключения из реестра. Затем проверяем параметр, переданный программе. Если программу запустили с ключом \s, то запрашиваем адрес сервера заново (возможно, он изменился), а не используем значение из реестра.

Далее все просто. Если необходимо, то выводим окно диалога для ввода IP-адреса и заносим результат в свойство `Address` компонента `SocketConnection`.

Так же как у сервера, у клиента не должно быть активных соединений при компиляции, а свойство `Address` я всегда очищаю. Ошибка при старте может возникнуть из-за активной таблицы или запроса.

Теперь на форму модуля данных добавляем два компонента: `DataSource` и `ClientDataSet`. Компонент `ClientDataSet` работает так же, как таблица `TTable`, потому что у них общий предок. Как и в случае компонента `TTable`, нужно связать компоненты `DataSource` и `ClientDataSet`, а вот соединение с базой данных настраивать не надо. Вместо этого в свойстве `RemoteServer` указываем компонент соединения с сервером *бизнес-логики* (в нашем случае — `stConnection`).

Так как у клиента не будет соединения с базой данных, то и не нужно указывать рабочую таблицу `TADOTable` или писать запрос в `TADOQuery`. Вместо этого нужно задать имя провайдера в свойстве `ProviderName`. Это имя компонента типа `TDataSetProvider`, который мы устанавливали в сервере бизнес-логики.

Установите на главную форму сетку `DBGrid`, чтобы увидеть данные. Теперь можно переходить к тестированию примера. Только предварительно не забудьте запустить утилиту bin\scktsrvr.exe из каталога установки Delphi.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\Levels\Client.

Выборка и изменение данных

Если вы запустите уже созданный пример и попробуете изменить данные, то ничего хорошего не выйдет. Проблема в том, что данные редактируются в кэше клиентского компьютера.

Чтобы измененные данные попали к серверу, необходимо выполнить метод `ApplyUpdates` компонента `ClientDataSet`. У этого метода один параметр — число возможных конфликтов, которые разрешено игнорировать. Что это значит? Допустим, два пользователя одновременно пытаются изменить одни и те же данные, какие из них сервер должен воспринять как верные и окончательные? Сервер не может вынести решение самостоятельно, поэтому будет генерироваться исключительная ситуация. Если количество таких ситуаций превысит значение, указанное в качестве параметра метода `ApplyUpdates`, то все изменения откатываются, и никакие данные не сохраняются.

В большинстве случаев вы будете вызывать метод `ApplyUpdates` с параметром 0, чтобы программа реагировала на любые ошибки. Но иногда может возникнуть ситуация, когда приходится сохранять хотя бы то, что не вызвало конфликтов.

Добавьте на форму клиентского приложения кнопку для сохранения результатов. Запустите приложение и внесите изменения в какую-либо строку (только не в названия улиц или городов, потому что эти поля связаны и попытка их изменения приведет к ошибке). После этого запустите программу SQL Profiler из состава MS SQL Server и посмотрите, какой запрос на обновление данных передается серверу MS SQL Server (листинг 3.18).

Листинг 3.18. Запрос на обновление

```
exec sp_executesql N'update tbClient  set
  vcName = @P1
where
  idClient = @P2 and
  vcName = @P3 and
  idContact is null and
  idTown = @P4 and
  idStreet = @P5 and
  cHouse is null and
  cOffice is null and
  idTown_1 = @P6 and
  vcTownName = @P7 and
  idStreet_1 = @P8 and
  idTown_2 = @P9 and
  vcStreetName = @P10
', N'@P1 varchar(51),@P2 varchar(39),@P3 varchar(51),@P4 varchar(39),@P5
   varchar(39),@P6 varchar(39),@P7 varchar(21),@P8 varchar(39),@P9
   varchar(39),@P10 varchar(51)', 'ПЛЮШКИ ИНТЕРНЕЙШНЛ 333',
   '{53EFC087-F90E-40EA-ADD8-555A9E995892}', 'ПЛЮШКИ ИНТЕРНЕЙШНЛ',
   '{F790F84C-4BB4-4654-8CAD-2A1355B67091}',
   '{2FC3656E-D271-46F1-975D-15209A00AF4D}',
   '{F790F84C-4BB4-4654-8CAD-2A1355B67091}', 'РОСТОВ-НА-ДОНУ',
   '{2FC3656E-D271-46F1-975D-15209A00AF4D}',
   '{F790F84C-4BB4-4654-8CAD-2A1355B67091}', 'БАРКО'
```

Неужели запрос на обновление одной только колонки может быть таким большим? Я бы записал все это в три строки и то только для того, чтобы удобнее было читать:

```
update tbClient
set vcName=Новое значение
WHERE idClient=ключевое поле
```

Такой запрос будет выполняться намного быстрее по двум причинам.

- Серверу базы данных легче разобрать этот запрос и сформировать план выполнения, потому что он гораздо проще и меньше.
- Так как в качестве критерия поиска указано ключевое поле, поиск пройдет достаточно быстро, потому что сервер сможет использовать индексы, а ключевые поля индексируются. Если ваш сервер автоматически не индексирует ключевые поля (если вы используете не MS SQL Server, а другой сервер баз данных), то сейчас же добавьте индекс вручную.

Так почему же запрос, который генерирует Delphi, такой большой? Ответ прост — при поиске предназначенной для изменения строки происходит сравнение

абсолютно всех полей. Программа сервера не знает про ключевое поле, по которому можно однозначно идентифицировать строку. Значит, необходимо помочь серверу.

Чтобы использовать ключи, возвращаемся к программе сервера бизнес-логики. Выделите компонент TDataSetProvider и в свойстве UpdateMode установите значение upWhereKeyOnly. После этого программа будет использовать ключи. Но нам надо еще указать, какое из полей является ключевым. Для этого дважды щелкните на компоненте TADOQuery, открыв уже знакомый нам редактор полей. Добавьте все поля. У ключевого поля в параметре pfInKey свойства ProviderFlag укажите значение true. Перекомпилируйте сервер.

Запустите приложение и посмотрите на генерируемый запрос. У меня он стал выглядеть так:

```
exec sp_executesql N'update tbClient  set
  vcName = @P1
where
  idClient = @P2
  ', N'@P1 varchar(50),@P2 uniqueidentifier', 'hoihjo',
  '53EFC087-F90E-40EA-ADD8-555A9E995892'
```

Вот это уже совсем другое дело. Здесь практически то же самое, что написал и я, но новое значение поля и значение ключа передаются через переменные (@P1 и @P2). Это даже лучше. Я тоже рекомендую все значения передавать серверу через переменные. Если выполнить два следующих запроса, то сервер воспримет каждый из запросов как новый и каждый запрос сервер будет разбирать отдельно, вырабатывая эффективный план выполнения, а это — лишние затраты времени:

```
update tbClient
set vcName='Новое имя'
WHERE idClient=ключевое поле

update tbClient
set vcName='Еще одно новое имя'
WHERE idClient=ключевое поле
```

Если передавать новое значение через переменную, то запросы будут считаться одинаковыми и план выполнения будет вырабатываться только для первого запроса, а для второго будет взят из кэша.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\ClientEditData.

Использование ключевого поля для обновления позволяет решить еще одну проблему. Допустим, выборка данных происходит из двух и более таблиц. Какую из таблиц обновлять при обновлении данных? При использовании двух уровней обновляется первая таблица в списке FROM. В трехуровневой системе по умолчанию для обновления используется запрос, в котором происходит поиск по всем полям. Такой запрос не сможет обновить данные из нескольких таблиц.

Если установить ключевое поле, то обновление заработает, хотя изменять можно будет только ту таблицу, ключевое поле которой мы указали, и это может быть только одна таблица.

При выполнении запросов сервер бизнес-логики производит кэширование данных. По умолчанию клиенту поступает 20 записей. Даже если выполнить запрос, результатом которого должно быть 100 записей, в свойстве RecordCount можно будет увидеть только число 20. При просмотре данных, например путем их прокрутки в сетке DBGrid, данные будут постепенно подгружаться, и для этого не нужно писать никакого кода.

Такая технология работы с данными позволяет снизить трафик и повысить скорость работы. Когда пользователи выполняют выборку данных, они нередко ошибаются в запросах. При поиске данных может понадобиться сделать несколько запросов к базе. Сервер передаст клиенту только первую партию данных, и в большинстве случаев их окажется достаточно для оценки, верно ли были заданы параметры поиска.

Если необходимо точно знать объем возвращаемых данных, я выясняю его перед выполнением запроса. Для этого секция SELECT в запросе переписывается следующим образом:

```
SELECT count(*)
```

Такой запрос выполняется достаточно быстро. Определив с его помощью объем возвращаемых данных, изменяем секцию SELECT так, чтобы можно было выбирать данные. Единственный недостаток такого метода — приходится выполнять два запроса: первый для определения объема возвращаемых данных и второй для выборки данных. В больших базах данных с множеством параллельно работающих пользователей это может стать серьезной проблемой. Между выполнением двух запросов на сервере может измениться многое. Например, может быть добавлена или удалена строка. В связи с этим полученное количество строк нужно воспринимать только как информационное значение, но никак не точное.

Если вы будете воспринимать полученный объем данных в качестве какой-то точки отсчета, то может возникнуть ошибка или результат выполнения команды окажется неточным. Например, пусть вам нужно выгрузить все данные в файл. Запрос для определения количества данных возвратил значения 1000 строк, но после этого другой пользователь удалил из базы данных 999 строк. Если взять за основу число 1000 и запустить цикл от 1 до 1000, в котором на каждой итерации одна строка сохраняется в файле, после чего происходит переход на следующую строку, то в файле окажутся неверные данные. Точнее сказать, единственная строка будет размножена 1000 раз, ведь переход на следующую строку (вызов метода Next) ничего не даст, поскольку в вашей выборке только одна строка и дальше переходить некуда.

Чтобы точно определить объем данных, необходимо программно переместиться в конец данных с помощью метода Last. В этом случае программе придется подгрузить все данные, чтобы получить последнюю строку. Такой подход требует больше времени, зато он безошибочен, а полученное значение уже не может измениться. Выбранные данные сохраняются в кэше, и работа происходит с данными

из кэша. Об изменении данных на сервере (их удалении или изменении) вы узнаете только после вызова метода ApplyUpdates, и только в этот момент начнутся ошибки.

Секреты работы с сервером бизнес-логики

Теперь рассмотрим некоторые секреты, связанные с сервером бизнес-логики. Тут есть несколько интересных нюансов. Мы не будем изучать их на практике, а ограничимся теорией. Что касается практики, то вам уже достаточно информации для воплощения в коде представленного в этой главе материала по управлению базами данных. Будем считать, что практическая часть остается вам в качестве домашнего задания для закрепления материала.

Реализация шаблонов

Реализация шаблонов для работы со справочниками в трехуровневой модели не сложна, потому что практически не требует изменений в коде. А вот при реализации окна для работы с клиентами возникнут проблемы, особенно при организации универсальных механизмов поиска и сортировки. Как сделать так, чтобы сервер бизнес-логики подставлял необходимые параметры в запрос? Можно предусмотреть переменные в запросе и заполнять их на стороне клиента, но таким образом мы потеряем универсальность и код не сможет работать с любым окном и любым набором данных.

Для реализации универсальных механизмов поиска и сортировки код запроса должен формироваться на стороне клиента. А как это сделать, если на стороне клиента используется компонент типа TClientDataSet, у которого нет свойства SQL и сложно изменить код запроса? Изменить сложно, зато заменить легко и просто.

У компонента TClientDataSet есть свойство CommandText. В него можно записать текст SQL-запроса, и сервер приложения обязан будет его выполнить. Но для этого на сервере в свойстве Options компонента TDataSetProvider должен быть установлен в true параметр poAllowCommandText.

И все же отсутствие свойства CommandText ведет к проблеме хранения запроса, ведь в нем все хранится в виде одной строки, а мы привыкли к многострочному тексту. Я решаю эту проблему следующим образом.

1. В объекте формы я создаю переменную SQL типа TStringList. В момент создания формы это свойство инициализируется и в него загружается шаблон запроса:
   ```
   SQL.LoadFromFile(имя файла);
   ```
2. Для создания запроса я работаю с переменной SQL. Так как тип переменной такой же, как и у свойства SQL компонента TADOQuery, адаптация кода не такая уж и сложная.
3. Перед тем как сделать компонент ClientDataSet активным, в его свойство я записываю содержимое переменной SQL. Это легко делается с помощью одной строки:
   ```
   ClientDataSet.CommandText:=SQL.Text;
   ```

Попробуйте реализовать универсальные поиск и сортировку описанным методом. Возможно, вы найдете какие-то оригинальные решения для улучшения кода. Окно универсального поиска и модуль UtilsUnit, адаптированные к трехуровневой модели, можно найти на компакт-диске в каталоге Sources\ch03\LevelsModules. Там находятся только модули, а реализацию я не стал включать в компакт-диск.

Давайте посмотрим, как будет происходить сортировка с использованием переменной SQL, чтобы вы лучше поняли новый алгоритм (листинг 3.19).

Листинг 3.19. Сортировка в трехуровневой модели

```
procedure TOrganizationForm.OrgDBGridTitleClick(Column: TColumn);
var
 SelectedIndex:Integer;
begin
SelectedIndex:=OrgDBGrid.SelectedField.Index;

 UniversalSortGrid(Column, sbOrganization.Panels[2], SortPointer,
      SortString, SQL);
 TClientDataSet(OrgDBGrid.DataSource.DataSet).Active:=false;
 TClientDataSet(OrgDBGrid.DataSource.DataSet).CommandText:=SQL.Text;
 TClientDataSet(OrgDBGrid.DataSource.DataSet).Active:=true;

 OrgDBGrid.SelectedIndex:=SelectedIndex-1;
end;
```

Обратите внимание, что в вызов функции UniversalSortGrid добавлен новый параметр. Через него передается переменная SQL, с помощью которой функция должна изменять раздел ORDER BY. После вызова функции полученный запрос заносится в свойство CommandText и выполняется.

Просмотр измененных данных

Теперь выясним, как можно просмотреть введенные пользователем изменения, прежде чем запомнить их в базе данных.

Итак, для реализации примера добавим в модуль данных компоненты DataSource и ClientDataSet. Их необходимо связать и указать в свойстве RemoteServer ваш сервер. В свойстве provider ничего указывать не надо, потому что никаких таблиц нам просматривать не нужно.

Теперь создадим форму, в которой будут отображаться изменения, введенные пользователем в таблицу. На этой форме нам понадобятся сетка DBGrid для отображения изменений и две кнопки — Откат и Отмена. По щелчку на кнопке Откат мы будем отменять последнее изменение, а по щелчку на кнопке Отмена — все изменения.

В обработчике событий OnClick кнопки Откат пишем следующий код:

```
ChangesDataSet.UndoLastChange(true);
UpdateDelta();
```

В первой строке вызывается метод UndoLastChange компонента ChangesDataSet, который отменяет последнее действие. Что за компонент ChangesDataSet? Это ука-

затель на компонент типа TClientDataSet, изменения в котором мы хотим увидеть. Значение этой переменной будет присваиваться при открытии окна.

Во второй строке вызывается метод UpdateDelta. В этом методе, представленном в листинге 3.20, мы обходим все поля набора данных DataSet, данные которого мы смотрим, и копируем из него содержимое свойств DisplayLabel и Visible в набор данных DataSet, отображающий изменения. Зачем это нужно? Мы не знаем, какую именно таблицу придется отображать, а ведь в ней могут быть ключевые поля, которые не должны отображаться, да и заголовки полей должны выглядеть так как надо. Именно для этого и задуман метод UpdateDelta.

Листинг 3.20. Обновление модуля данных

```
procedure TChangesForm.UpdateDelta;
var
  i:Integer;
begin
   DataModule1.tbChanges.Data:=ChangesDataSet.Delta;
   for i:=0 to DataModule1.tbChanges.FieldCount-1 do
    begin
     DataModule1.tbChanges.Fields[i].DisplayLabel:=
               ChangesDataSet.Fields[i].DisplayLabel;
     DataModule1.tbChanges.Fields[i].Visible:=
               ChangesDataSet.Fields[i].Visible;
    end;
end;
```

А теперь самое интересное — отображение окна. Для этого напишем метод ShowDelta, представленный в листинге 3.21.

Листинг 3.21. Метод отображения окна с изменениями

```
procedure TChangesForm.ShowDelta(ChangesSet: TClientDataSet);
begin
  try
   ChangesDataSet:=ChangesSet;
   UpdateDelta();

   ShowModal;
  except
   Application.MessageBox('В журнале нет изменений', 'Ошибка',
     MB_OK+MB_ICONERROR);
   exit;
  end;
end;
```

Методу нужно передать только указатель на компонент типа TClientDataSet. В первой строке кода мы сохраняем этот указатель. Затем вызываем метод UpdateDelta, в котором обновляется содержимое окна изменений. Если в этот момент происходит ошибка, значит, изменений не было.

Все готово. Чтобы отобразить окно просмотра изменений (рис. 3.13) для таблицы клиентов достаточно выполнить следующую строку:

```
ChangesForm.ShowDelta(DataModule1.tbdClients);
```

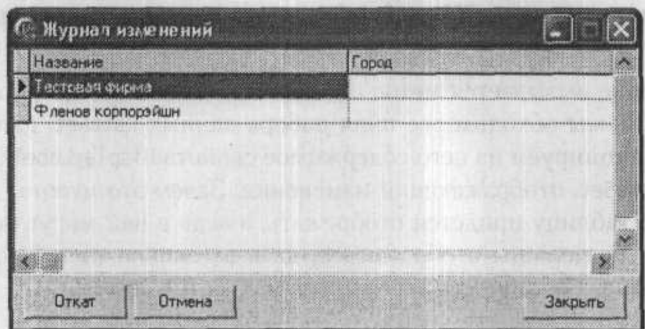


Рис. 3.13. Окно просмотра изменений

Чтобы увидеть изменения другой таблицы, необходимо указать ее в качестве параметра. Таким образом, у нас получилось окно, способное отражать изменения любого набора данных.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\Changes.

Обработка ошибок сохранения

При щелчке на кнопке сохранения данных вызывается метод ApplyUpdates. Если во время сохранения произойдет ошибка, данные записаны не будут, а (самое страшное!) пользователь об этом ничего не узнает, и изменения окажутся потерянными. Чтобы пользователь мог реагировать на ошибки, в программу нужно встроить обработчик этих ошибок.

Писать собственный обработчик достаточно утомительно. Намного проще взять за основу уже готовые функции и формы. Сначала рассмотрим, как можно воспользоваться встроенными в Delphi механизмами. Обработчик события OnReconcileError будем создавать для компонента TClientDataSet. Здесь достаточно написать одну строку:

```
Action:=HandleReconcileError(DataSet, UpdateKind, E);
```

Функция HandleReconcileError, объявленная в модуле recerror, в случае ошибки выводит окно с подробной информацией об ошибке и конфликтующих полях. В этом окне пользователь может выбрать дальнейшие действия: проигнорировать ошибку, откорректировать ошибку, обновить данные или прервать процесс обновления.

Окно, которое нам предоставляет Delphi, достаточно удобное и информативное, но только для тех, кто знает английский язык. Чтобы не переписывать весь код заново, я подкорректировал код окна и сохранил его в модуле UpdateConflictUnit. Теперь, если мне нужно русскоязычное окно, я просто помещаю этот модуль в папку проекта, и Delphi автоматически начинает использовать мою версию окна разрешения конфликтов (рис. 3.14), а не встроенную.

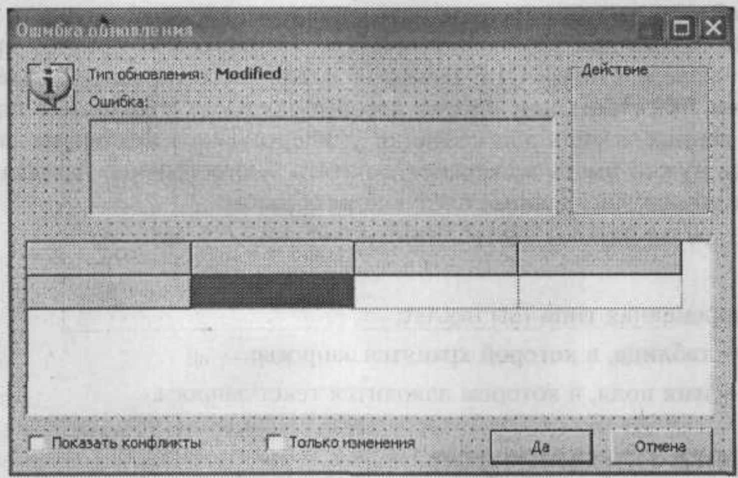


Рис. 3.14. Окно разрешения конфликтов

В принципе, код модуля `UpdateConflictUnit` не такой уж сложный и, немного постаравшись, его можно переписать полностью за один день. Но зачем это делать, если уже все готово? Просто украсим окно и сделаем его более удобным для русскоязычного населения.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\ApplyData.

Сторонние разработки

Я всегда предлагаю свои варианты решений, но никогда не утверждаю, что только они являются единственно верными. Сейчас мы рассмотрим другие варианты. Я позаимствовал их у своих друзей. Возможно, они помогут вам, и реализованные в них идеи вы воплотите в жизнь в своих проектах.

Хранение запросов

Я рекомендовал вам хранить запросы в файле. Это действительно удобно, потому что простые изменения можно вносить в программу без ее перекомпиляции, только путем изменения запросов. Так намного проще обновлять программу и на компьютерах пользователей. Если вы нашли в своем запросе ошибку, для ее исправления достаточно будет обновить текстовый файл запроса.

Очень часто у некоторых пользователей программа должна по-разному отображать данные. Например, бухгалтер, просматривая проводки, должен видеть все, в то время как экономисту нужна только касающаяся его работы узкоспециализированная информация. Эту проблему легко решить двумя текстовыми файлами с запросами — один будет загружаться для бухгалтера, другой для экономиста.

Однако файлы — не единственное место, где можно хранить запросы. Я использовал файл только для упрощения примера. Намного эффективнее хранить запросы на сервере в виде OLE-объектов и для редактирования задействовать компоненты TDBRichEdit или TDBMemo, которые позволят сохранять в поле многострочные данные, а ведь для создания универсального механизма поиска или сортировки нужно иметь возможность писать многострочные запросы. При загрузке нужно получать данные следующим образом:

```
SQL.Text := DataModule1.Table1.FieldByName('Запрос').AsString
```

Здесь:

- SQL — переменная типа TStringList;
- Table1 — таблица, в которой хранятся запросы;
- Запрос — имя поля, в котором находится текст запроса.

Передача параметров

Как реализовать передачу параметров, если у вас запрос храниться в файле или в таблице на сервере? Как это сделать в отношении хранимых процедур? Рассмотрим наш шаблон запроса:

```
SELECT *
FROM tbClient cl, tbTown t, tbStreet s
WHERE cl.idTown*=t.idTown AND cl.idStreet*=s.idStreet
```

Допустим, имя клиента вы хотите искать через параметр:

```
SELECT *
FROM tbClient cl, tbTown t, tbStreet s
WHERE cl.idTown*=t.idTown AND cl.idStreet*=s.idStreet
      AND cl.vcName LIKE :ClientName
```

В последней строке поле vcName таблицы клиентов сравнивается с параметром ClientName, который необходимо изменять перед выполнением запроса. Как реализовать замену? Допустим, запрос загружен в поле SQL компонента ADOQuery1, тогда проблема решается одной строкой:

```
ADOQuery1.SQL.Text := StringReplace(
  ADOQuery1.SQL.Text,
  ':ClientName',
  Edit1.Text,
  [rfReplaceAll]);
```

Функции StringReplace передаются четыре параметра:

- переменная, в которой нужно найти и изменить какой-то текст;
- искомый текст;
- текст для замены;
- флаг (в данном случае указан флаг rfReplaceAll, который означает, что нужно заменить все найденные вхождения текста из второго параметра).

Функция возвращает измененную строку, поэтому сохраняем результат в переменной Text запроса.

Единственная проблема этого метода состоит в том, что после первого выполнения команды ее повторное выполнение оказывается невозможным, потому что

параметра ClientName больше нет в запросе. Команда ничего не найдет и ничего не заменит, поэтому придется перезагружать запрос из файла, что не всегда удобно.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\5-Parameters.

Создание универсальных окон

Есть еще один вариант создания универсальных окон для работы с таблицами базы данных. Его я видел у двух своих знакомых, но использовать в своих проектах так и не решился. Создается окно, в котором каким-то образом сохраняются три запроса: на выборку данных, на сохранение изменений в таблице и на удаление строк. Помимо этого необходимо форме сообщить типы и имена полей, которые должны отображаться на форме, если не используются отдельные компоненты TADOQuery для каждого запроса.

Преимущество метода состоит в том, что добавление, изменение и удаление записей происходит по запросам. Иногда это действительно полезно. В описанном ранее примере за изменение отвечает компонент TADOQuery, что очень удобно, но неэффективно при редактировании связанных таблиц.

В моем варианте, описанном ранее в этой главе, для каждого справочника создавался отдельный экземпляр TADOTable, а здесь вне зависимости от таблицы достаточно максимум двух для всех справочников, которые будут выполнять необходимые запросы.

В реализации этот метод более сложен, потому что труднее создать удобное для работы пользователя окно. Слишком большая степень универсальности всегда приводит к множеству ограничений и неудобств.

Я не буду реализовывать этот пример в данной книге, потому что вижу в нем очень много проблемных мест и не вижу элегантного решения. Создавать некрасивые решения можно только для собственных сугубо личных проектов, а выставлять их на показ не очень хочется. В какой-то степени даже стыдно. Тем не менее умолчать об этом методе я тоже не мог, потому что всегда стараюсь дать читателям выбор. А какой выбор сделаете вы, зависит только от личных предпочтений и от конкретной ситуации.

Косметика

В этой главе я собрал наиболее интересные решения, которые сам не раз уже использовал в своих программах, надеюсь, что они пригодятся и вам. Все примеры основаны на личном опыте создания программ для конечного пользователя и направлены на упрощение разработки программ и их использования.

Выделение строк с помощью закладок

Для начала посмотрим, как можно выделять записи в сетке DBGrid. У этого компонента есть параметр MultiSelect, но работать с ним достаточно неудобно. Дело в том, что выделять строки приходится с помощью клавиш со стрелками, удерживая клавишу Shift. Чтобы выделить несмежные строки, без мыши не обойтись, а это пользователи не очень любят. Я их прекрасно понимаю, ведь использовать для этих целей клавишу пробела намного удобнее.

Однажды я заглянул в исходный код сетки, в которой была реализована нужная мне функциональность, и просто испугался чрезмерной сложности кода. Жаль бедных программистов, которые написали столько кода, а еще больше жаль тех, кто будет вынужден этот код сопровождать. Возможно, разработчики действительно видели необходимость в таком усложнении, но я такой необходимости не заметил.

Итак, давайте создадим приложение, в котором записи в сетке можно будет выделять с помощью клавиши пробела. Как все это реализовать? Есть два варианта простого решения проблемы:

- создать список и при выделении записи сохранять в списке ключевое поле текущей строки;
- создать список и при выделении записи сохранять закладку на текущую запись.

Как видите, оба решения примерно одинаковы и основаны на списке. Разница только в том, что именно хранится в этом списке. Мне больше нравится второй вариант, потому что он удобнее, когда нужно пересмотреть все выделенные записи.

Сетка DBGrid предлагает два варианта закладок — в виде строк и в виде указателей. В данном случае наилучшим вариантом будут указатели, для хранения которых удобно использовать список TList. Итак, в разделе private объявления нашей формы создаем переменную SelectedList типа TList:

```
SelectedList:TList;
```

Инициализировать эту переменную нужно по событию OnCreate или OnShow:

```
SelectedList:=TList.Create;
```

Если следовать правилам хорошего тона, то по событию OnClose нужно уничтожать эту переменную методом SelectedList.Free.

А вот теперь самое интересное — выделение. Для этого создаем обработчик события OnKeyDown. Код этого обработчика можно увидеть в листинге 3.22.

Листинг 3.22. Выделение записи

```
procedure TChildTemplateForm.dbgMainGridKeyDown(
  Sender: TObject; var Key: Word; Shift: TShiftState);
var
 i: Integer;
 bm:Pointer;
begin
 // Если нажата клавиша пробела, то
 if Key=VK_SPACE then
  begin
   // Получаем закладку
```

```
    bm:=TDataSet(dbgMainGrid.DataSource.DataSet).GetBookmark;

    // Запускаем цикл
    for i:=0 to SelectedList.Count-1 do
     // Если закладка существует в списке, удаляем ее
     if TDataSet(dbgMainGrid.DataSource.DataSet).CompareBookmarks(
         SelectedList[i],
         bm)=0 then
      begin
       SelectedList.Delete(i);
       TDataSet(dbgMainGrid.DataSource.DataSet).Next;
       exit;
      end;

    // Добавляем закладку
    SelectedList.Add(bm);
    TDataSet(dbgMainGrid.DataSource.DataSet).Next;
   end;
 end;
```

Что здесь происходит? Если нажата клавиша пробела, то в переменной `bm` сохраняется закладка на текущую строку.

Затем мы должны убедиться, что в списке нет закладки на эту строку. Если есть, то выделение должно быть снято. Получается, что если первый раз пользователь нажимает клавишу пробела, строка выделяется, а второе нажатие снимает выделение.

Для поиска закладки в списке запускается цикл, в котором все элементы перебираются и сравниваются с текущей закладкой. Хотя закладки имеют тип `Pointer`, их нельзя сравнивать с помощью простого знака равенства. Необходимо использовать метод `CompareBookmarks`, которому передаются два указателя. Если закладки равны, то метод возвращает 0.

Если закладка на текущую строку найдена, то удаляем ее из списка, переходим на следующую строку и выходим из метода:

```
SelectedList.Delete(i);
TDataSet(dbgMainGrid.DataSource.DataSet).Next;
exit;
```

Если цикл завершился и начал выполняться следующий за циклом код, значит, текущая запись не была выделена. Выделяем. Для этого добавляем закладку в список и переходим на следующую запись:

```
SelectedList.Add(bm);
TDataSet(dbgMainGrid.DataSource.DataSet).Next;
```

Переход на следующую строку не является обязательным, но он очень удобен, когда пользователь хочет выбрать сразу несколько смежных строк. Достаточно просто нажимать клавишу пробела, не используя клавиши со стрелками или мышь.

Итак, у нас есть список для хранения закладок для выделенных строк, но он пока не полностью функционален. Дело в том, что пользователь не будет видеть на экране, какие строки выделены, а какие нет. Отображение придется реализовывать самостоятельно, но дело того стоит, да и не так уж это и сложно. Отображение будет происходить в обработчике события `OnDrawDataCell` (листинг 3.23).

Листинг 3.23. Отображение выделенных строк

```
procedure TChildTemplateForm.dbgMainGridDrawDataCell(Sender: TObject;
  const Rect: TRect; Field: TField; State: TGridDrawState);
var
 i:Integer;
 bm:Pointer;
begin
 // Определяем закладку
 bm:=TDataSet(dbgMainGrid.DataSource.DataSet).GetBookmark;

 // Запускаем цикл
 for i:=0 to SelectedList.Count-1 do
  // Если срока есть в списке, то меняем цвет фона
  if TDataSet(dbgMainGrid.DataSource.DataSet).CompareBookmarks(
       SelectedList[i], bm)=0 then
   begin
    dbgMainGrid.Canvas.Brush.Color:=clHighlight;
   end;

 // Прорисовываем ячейку
 dbgMainGrid.Canvas.FillRect(Rect);
 dbgMainGrid.Canvas.TextOut(Rect.Left+2, Rect.Top+2, Field.AsString);
end;
```

Логика метода достаточно проста — нужно перебрать весь список, и если текущая строка в нем есть, то запись должна быть выделена, а значит, устанавливаем цвет фона (`Brush`) в цвет выделения `clHighlight`. После этого закрашиваем текущую ячейку сетки цветом фона и выводим текст ячейки.

Почему я определяю закладку до начала цикла? Это сделано в целях повышения производительности. Закладка не меняется, она равна текущей строке, и смысла определять ее на каждой итерации нет. Достаточно один раз занести ее значение в локальную переменную и использовать. При большом количестве выделенных строк этот метод может реально повысить производительность.

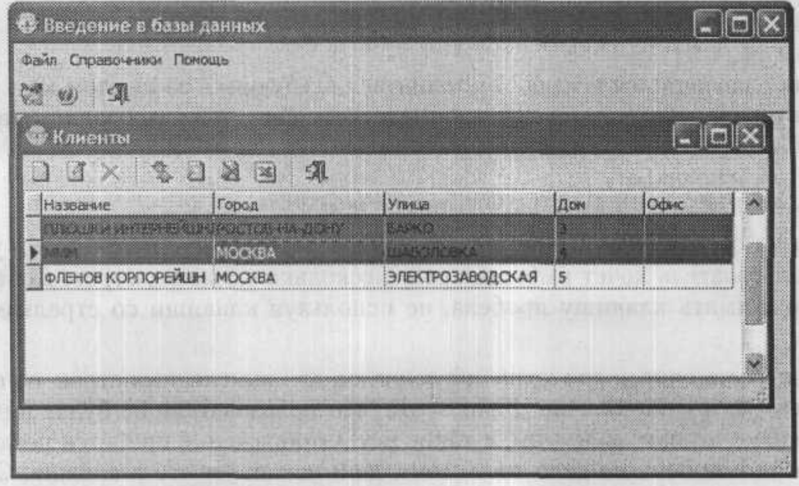


Рис. 3.15. Результат работы программы

Запустите программу и попробуйте выделить строки с помощью клавиши пробела (рис. 3.15). Если вы сделаете это, то увидите, что данный метод выделения намного удобнее стандартного.

Теперь посмотрим, как можно использовать выделенные строки для выполнения групповых операций. Например, нужно удалить все выделенные строки. Для этого выполняем следующий код:

```
var
 i: Integer;
begin
 with TDataSet(dbgMainGrid.DataSource.DataSet) do
  for i:=0 to SelectedList.Count-1 do
   begin
    if not BookmarkValid(SelectedList[i]) then
     continue;
    GotoBookmark(SelectedList[i]);
    // Удалить запись
   end;
 // Очистить список выделенных записей
end;
```

Чтобы код был проще и чтобы постоянно не приходилось писать длинную строку `TDataSet(dbgMainGrid.DataSource.DataSet)`, я выполняю оператор `with`.

Далее запускается цикл. Здесь мы сначала должны убедиться, что ссылка еще действительна. Для проверки используется метод `BookmarkValid`. Если метод возвращает ложное значение, то текущая закладка не используется, а выполняется переход к следующему элементу в списке. Если закладка действительна, то выполняется переход на соответствующую запись. Для этого используется метод `GotoBookmark`. После этого можно выполнять необходимые действия с записью.

Если выполнялось удаление, то по завершении цикла список можно очистить; так как соответствующих записей уже нет в таблице, все закладки в списке становятся ошибочными. Для очистки достаточно использовать метод `Clear`:

```
SelectedList.Clear
```

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\select1.

Выделение строк через ключ

Теперь рассмотрим еще один метод выделения — через ключевое поле. Этот метод лучше подходит, если у вас реализована сортировка. Попробуйте запустить приложение из предыдущего раздела, выделить несколько строк и выполнить сортировку. После завершения сортировки окажутся выделенными не те поля, которые выделяли вы. Эта проблема закладок достаточно легко решается с помощью ключевых полей.

Итак, ключ — это не ссылка, но чаще всего ключевое поле имеет тип, который легко привести к строковому (автоматически увеличиваемое поле или глобально уникальный идентификатор). Значит, для хранения выделенных строк подойдет

список строк TStringList. Меняем тип переменной SelectedIndex и строку инициализации переменной в обработчике событий OnShow:

```
SelectedList:=TStringList.Create;
```

Теперь модифицируем обработчик события OnKeyDown, как показано в листинге 3.24. Далее нам надо сохранять в списке ключевое поле. Для простоты я всегда оставляю ключевое поле на нулевой позиции, то есть первым в списке.

Листинг 3.24. Обработчик события OnKeyDown

```
procedure TChildTemplateForm.dbgMainGridKeyDown(
  Sender: TObject; var Key: Word; Shift: TShiftState);
var
 i: Integer;
 bm: String;
begin
 // Если нажата клавиша пробела, то
 if Key=VK_SPACE then
  begin
   // Определяем ключ строки
   bm:=TDataSet(dbgMainGrid.DataSource.DataSet).Fields[0].AsString;

   // Запускаем цикл
   for i:=0 to SelectedList.Count-1 do
    // Если строка есть в списке, то снимаем выделение
    if SelectedList[i] = bm then
     begin
      SelectedList.Delete(i);
      TDataSet(dbgMainGrid.DataSource.DataSet).Next;
      exit;
     end;

   // Добавляем строку
   SelectedList.Add(bm);
   TDataSet(dbgMainGrid.DataSource.DataSet).Next;
  end;
end;
```

Теперь посмотрим на новый метод прорисовки (листинг 3.25).

Листинг 3.25. Обработчик события OnDrawDataCell

```
procedure TChildTemplateForm.dbgMainGridDrawDataCell(Sender: TObject;
  const Rect: TRect; Field: TField; State: TGridDrawState);
var
 i:Integer;
 bm:String;
begin
 // Определяем текущий ключ
 bm:=TDataSet(dbgMainGrid.DataSource.DataSet).Fields[0].AsString;
 for i:=0 to SelectedList.Count-1 do
  // Если срока есть в списке, то меняем цвет фона
  if SelectedList[i]= bm then
   begin
    dbgMainGrid.Canvas.Brush.Color:=clHighlight;
   end;
```

```
  // Прорисовываем ячейку
  dbgMainGrid.Canvas.FillRect(Rect);
  dbgMainGrid.Canvas.TextOut(Rect.Left+2, Rect.Top+2, Field.AsString);
end;
```

Как видите, разница не такая уж и большая по сравнению с примером, реализующим выделение с помощью закладок. Максимальное отличие проявляется при выполнении групповых операций с выделенными строками. Давайте посмотрим, как это происходит:

```
var
 i: Integer;
begin
 with TDataSet(dbgMainGrid.DataSource.DataSet) do
  for i:=0 to SelectedList.Count-1 do
   begin
    dbgMainGrid.DataSource.DataSet.Locate(
      dbgMainGrid.DataSource.DataSet.Fields[0].FieldName,
      SelectedList[i], [loCaseInsensitive]);
    // Удалить запись
   end;
 // Очистить  список выделенных записей
end;
```

Чтобы быстро перейти к строке, зная ее ключ, лучше всего использовать метод Locate, которому нужно передать три параметра:

- имя ключевого поля — в данном примере мы передаем в этом параметре свойство FieldName нулевого поля;
- значение ключевого поля — мы передаем *i*-й элемент из списка выделенных строк;
- параметры поиска — для ключа параметром [loCaseInsensitive] я задаю нечувствительность к регистру символов (если ключевое поле является числом или GUID, а не строкой, параметры можно не указывать).

Если все указано верно, метод Locate представит нам строку с указанным ключом. Не помешает проверить, что выделенная строка действительно является той строкой, которая нам нужна, поскольку в случае ее удаления метод Locate выделит первую попавшуюся запись. Проверку лучше выполнить после вызова метода Locate:

```
if OrgDBGrid.DataSource.DataSet.Fields[0].AsStrin<>
  SelectedList[i] then
  Ошибка позиционирования
```

Вот вроде бы и все, что необходимо знать о выделении. Теперь вы сможете расширить возможности своей программы и сделать групповые операции более удобными.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\select2.

Позиционирование

Метод Locate очень удобен для быстрого поиска строки в выборке данных. Например, если в нашей программе после сортировки текущей становится первая строка в выборке, проблему можно исправить, до сортировки запоминая ключ выделенной строки, а после сортировки находя строку по этому ключу. Модифицированный метод сортировки с учетом сохранения текущей строки можно увидеть в листинге 3.26.

Листинг 3.26. Сортировка с сохранением текущей позиции

```
procedure TChildTemplateForm.dbgMainGridTitleClick(Column: TColumn);
var
 Index: Integer;
 sKey:String;
begin
 // Запоминаем выделенную строку
 sKey:=TDataSet(dbgMainGrid.DataSource.DataSet).Fields[0].AsString;

 // Запоминаем текущую выделенную колонку
 Index:=dbgMainGrid.SelectedIndex;
 // Функция генерации строки сортировки
 UniversalSortGrid(Column,
    StatusBar1.Panels[0], SortPointer, SortString,
    TADOQuery(dbgMainGrid.DataSource.DataSet).SQL);

 // Перечитываем данные
 TADOQuery(dbgMainGrid.DataSource.DataSet).Active:=false;
 TADOQuery(dbgMainGrid.DataSource.DataSet).Active:=true;

 // Восстанавливаем выделенную колонку
 dbgMainGrid.SelectedIndex:=Index;

 // Восстанавливаем выделенную строку
 dbgMainGrid.DataSource.DataSet.Locate(
      dbgMainGrid.DataSource.DataSet.Fields[0].FieldName,
      sKey, [loCaseInsensitive]);
end;
```

Позиционировать курсор можно не только по ключевому полю, но и по любому другому. Например, чтобы найти запись, в которой название организации равно «ммм», можно не выполнять отдельного запроса, а просто позиционировать курсор на соответствующую запись в существующей выборке, используя метод Locate. Например, так:

```
procedure TChildTemplateForm.Button1Click(Sender: TObject);
begin
 dbgMainGrid.DataSource.DataSet.Locate(
    'vcName',
    'ммм', [loCaseInsensitive]);
end;
```

В качестве первого параметра мы передаем имя поля 'vcName', которое не является ключевым, но поиск по нему возможен. Второй параметр передает искомое значение. Так как название организации — это строка, то каждый пользователь может написать ее по-своему, используя прописные или строчные буквы. Чтобы

результат поиска не зависел от регистра символов, в третьем параметре передается значение loCaseInsensitive.

А что произойдет, если запись найти не удастся? В этом случае метод возвращает ложное значение. Конечно же, можно проверять результат, сравнивая значение поля текущей строки с искомым, как мы это делали в предыдущем разделе, а можно выводить сообщение об ошибке. В следующем примере если метод Locate не находит строку, то на экране появляется сообщение об ошибке:

```
procedure TChildTemplateForm.Button1Click(Sender: TObject);
begin
  if not dbgMainGrid.DataSource.DataSet.Locate(
     'vcName', 'мммw', [loCaseInsensitive]) then
   ShowMessage('Ошибка, запись не найдена');
end;
```

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\position.

Экспорт в Word

В книге «Библия Delphi» показано, как можно экспортировать данные в Excel, однако упущена тема экспорта в Word, а судя по письмам читателей, эта тема интересует очень многих. Теперь я решил исправить ошибку.

Если вы читали книгу «Библия Delphi» или просто работали с COM-объектами, то принцип взаимодействия с Word покажется вам знакомым.

Создание документа Word

Создадим новое приложение и сразу же добавим в раздел uses модуль ComObj, который содержит все необходимые для тестирования приложения методы. Затем поместим на главной форме кнопку, по щелчку на которой будет выполняться код из листинга 3.27. Этот код запускает Word, создает пустой документ и печатает тестовое сообщение.

Листинг 3.27. Тест приложения Word

```
procedure TForm1.bnTestWordClick(Sender: TObject);
var
 WordApp, Doc: Variant;
begin
 try
  WordApp:= CreateOleObject('Word.Application');
 except
  ShowMessage('Похоже, что Word не установлен :(');
  exit;
 end;
 WordApp.Visible:=true;
 Doc:=WordApp.Documents.Add;
 WordApp.Selection.TypeText('Это тестовое сообщение');
end;
```

Word можно запустить с помощью метода CreateOleObject, которому передается имя COM-объекта. Для Word этим именем является Word.Application. Если приложение запустить не удается, генерируется исключение. Это может означать, что программа не установлена или неработоспособна. Если Word удается запустить вручную, то программно (с помощью метода CreateOleObject) это тоже должно получаться.

Результат выполнения функции CreateOleObject записывается в переменную WordApp. Через эту переменную можно управлять приложением Word. В данном примере для иллюстрации мы сначала изменяем свойство Visible на true. Это свойство отвечает за видимость приложения, а по умолчанию оно невидимо, но доступно.

Если окно Word присутствует на экране, то любые изменения, которые мы ему направляем, будут тут же отображаться в окне. Это достаточно неудобно и отрицательно влияет на производительность. Допустим, вы выводите на экран посимвольно 1000 символов. После вывода каждого символа окно Word будет перерисовываться. Если же окно невидимо, то и перерисовываться нечему, а значит, не будет лишних операций.

Поэтому при выводе больших объемов данных я рекомендую делать окно Word невидимым (устанавливать свойство Visible в false) и возвращать ему видимость только после заполнения окна Word данными. В этом случае при внесении 1000 изменений в данные документа мы экономим 1000 операций прорисовки.

Для иллюстрации в нашем примере я сделал окно Word видимым в самом начале, но в реальных приложениях лучше это делать в самом конце кода, когда документ уже сформирован (если же вообще оставить окно Word невидимым, то реально оно на экране не появится, хотя в списке запущенных приложений вы сможете найти файл WINWORD.EXE).

Еще одна причина, по которой свойство Visible нужно устанавливать в true только в самом конце кода, — защита от сбоев. Дело в том, что в момент формирования документа пользователь может поменять положение курсора или щелкнуть на кнопке панели инструментов, и это может привести к сбою в программе, тогда дальнейшее формирование документа станет невозможным. То есть документ, который заполняется программно, не должен изменяться вручную, пока программа не завершит работу.

Следующим этапом я создаю новый документ. Для работы с документами есть свойство Documents, а для создания нового документа у этого свойства есть метод Add. Итак, для создания документа нужно вызвать метод WordApp.Documents.Add. Результат помещаем в переменную Doc типа Variant, чтобы можно было впоследствии управлять этим документом.

Вывод текста

Для работы с текстом документа используется свойство Selection, указывающее на выделение или текущую позицию курсора в документе. Для того чтобы заставить Word напечатать текст в текущей позиции, вызываем метод TypeText свойства Selection. В примере таким образом печатается тестовое сообщение:

```
WordApp.Selection.TypeText('Это тестовое сообщение');
```

Запустите программу и убедитесь, что все работает верно.

Чтобы программно закрыть документ, нужно использовать метод Close созданного документа. В нашем случае на документ указывает переменная Doc, а значит, команда закрытия будет выглядеть следующим образом:

```
Doc.Close;
```

В примере я не закрывал документ, чтобы вы могли увидеть результат работы после выполнения программы.

У некоторых программистов вызывает трудности создание в документе новой строки. Первое, что приходит в голову, — передать в Word команду печати символов конца строки и перевода каретки (с кодами #13 и #10). Если хотите, можете так и сделать. В следующем примере вводим два текстовых сообщения, каждое из которых будет находиться в отдельной строке:

```
WordApp.Selection.TypeText('Это тестовое сообщение'+#13+#10);
WordApp.Selection.TypeText('Это тестовое сообщение');
```

В конец первого сообщения мы добавляем символы конца строки и перевода каретки, чтобы текст следующего сообщения программа начала печатать с начала строки.

Однако в MS Word есть более красивый метод решения проблемы перехода на следующую строку — это метод TypeParagraph свойства Selection. В следующем примере также печатаем два сообщения, каждое из которых будет расположено в отдельной строке:

```
WordApp.Selection.TypeText('Это тестовое сообщение');
WordApp.Selection.TypeParagraph;
WordApp.Selection.TypeText('Это тестовое сообщение');
```

Позиционирование

При выводе очередного сообщения курсор в программе MS Word смещается в самый конец текстового блока. А что если нам нужно переместить курсор в другую позицию, например вернуться назад? Допустим, мы напечатали сообщение «Это тестовое сообщение», и теперь курсор находится в конце слова «сообщение». Как вставить слово «текстовое» после слова «тестовое»? Для этого нужно переместить курсор на девять символов назад и напечатать в этой позиции нужное слово.

Для перемещения курсора используется метод Move свойства Selection. У этого метода два параметра: точность движения и количество шагов. Количество шагов зависит от точности, но что это такое? Соответствующие числовые значения и их описания представлены в табл. 3.1.

Таким образом, чтобы переместить курсор на пять символов, нужно указать в качестве точности значение 1, а в качестве числа шагов — значение 5. Количество шагов может быть и отрицательным. Если это значение положительное, то курсор перемещается вправо, если отрицательное — влево. При этом имейте в виду, что во время движения курсора происходит выделение, примерно так

же, как в случае перемещения курсора с помощью клавиш со стрелками в программе Word при нажатой клавише Shift.

Таблица 3.1. Таблица значений точности перемещения курсора

Точность	Шаг
1	Символ
2	Слово
3	Предложение
4	Абзац
5	Строка
9	Столбец таблицы
10	Строка таблицы
12	Ячейка таблицы

В следующем примере вставляем слово «текстовое» перед словом «сообщение»:

```
WordApp.Selection.TypeText('Это тестовое сообщение');
WordApp.Selection.Move(1, -9);
WordApp.Selection.TypeText('текстовое сообщение');
```

Метод Move достаточно прост. Более мощными являются его варианты MoveLeft и MoveRight. Первый из этих методов двигает курсор влево, второй — вправо. У обоих методов три параметра:

- точность;
- количество шагов;
- необходимость выделения.

Если третий параметр равен true, то текст выделяется, иначе просто движется курсор. Таким образом, задача вставки слова может быть решена следующим образом:

```
WordApp.Selection.TypeText('Это тестовое сообщение');
WordApp.Selection.MoveLeft(1, 9, false);
WordApp.Selection.TypeText('текстовое ');
```

Если необходимо именно выделить текст, то в качестве третьего параметра метода MoveLeft передаем true. Чтобы снять выделение, вызывается метод Collapse:

```
WordApp.Selection.Collapse;
```

Где окажется курсор после выполнения метода Collapse? Там, куда мы его сдвинули. Пример:

```
// Печатаем текст
WordApp.Selection.TypeText('Это тестовое сообщение');
// Сдвигаем текст влево с выделением
// Выделенным будет слово "сообщение"
WordApp.Selection.MoveLeft(1, 9, true);
// Убираем выделение, курсор будет перед словом "сообщение"
WordApp.Selection.Collapse;
// Вставляем слово
WordApp.Selection.TypeText('текстовое ');
```

Теперь посмотрим, как можно подняться на одну строку выше. Для этого в качестве точности нужно указать число 5 (первый параметр метода Move). В следующем примере мы поднимаемся на одну строку вверх:

```
WordApp.Selection.Move(5, -1);
```

Попробуйте сами проверить остальные параметры точности (только имейте в виду, что не все параметры могут использоваться с методом MoveLeft или MoveRight).

Шрифт

Очень трудно читать отчет, в котором весь текст набран одним шрифтом. В отчете есть как минимум заголовок, который всегда можно выделить жирным шрифтом.

Для управления шрифтами используется свойство Font сложного свойства Selection. Параметры, которые можно изменять у шрифта, перечислены в таблице 3.2.

Таблица 3.2. Параметры шрифта

Название параметра	Тип	Описание
Name	Строка	Позволяет изменить шрифт. Параметру присваивается имя требуемого шрифта
Size	Число	Число, определяющее размер шрифта
Bold	Булев	Если параметр равен true, то текст будет печататься жирным шрифтом
Italic	Булев	Если параметр равен true, то текст будет печататься курсивом
StrikeThrough	Булев	Если параметр равен true, то текст будет печататься перечеркнутым
Subscript	Булев	Если параметр равен true, то текст будет печататься в нижнем индексе
Superscript	Булев	Если параметр равен true, то текст будет печататься в верхнем индексе
SmallCaps	Булев	Если параметр равен true, то текст будет печататься малыми прописными буквами
AllCaps	Булев	Если параметр равен true, то текст будет печататься большими прописными буквами

В следующем примере текст печатается шрифтом Courier:

```
WordApp.Selection.Font.Name:='Courier';
WordApp.Selection.TypeText('Это тестовое сообщение');
```

Теперь попробуем напечатать в документе число 10 и указать для него вторую степень. Для этого нужно напечатать число 10, потом установить свойство Selection.Font.Superscript в true и напечатать 2. Не забудьте вернуть шрифт в исходное состояние:

```
WordApp.Selection.TypeText('10');
WordApp.Selection.Font.Superscript:=true;
WordApp.Selection.TypeText('2');
WordApp.Selection.Font.Superscript:=false;
```

Размещение текста

Теперь нам предстоит познакомиться с размещением текста на листе. Для этого у MS Word есть достаточно широкие возможности, и все они доступны для изменения программно. Так как возможностей много, то рассматривать их все мы не будем, а вот основные возможности обязательно изучим.

Следующие параметры влияют на размещение текста и относятся к свойству `Paragraphs`:

- `FirstLineIndent` — отступ для первой строки абзаца (красная строка);
- `LeftIndent` — отступ слева для всех строк абзаца;
- `RightIndent` — отступ справа для всех строк абзаца;
- `Alignment` — выравнивание:
  - 0 — по левому краю;
  - 1 — по центру;
  - 2 — по правому краю;
  - 3 — по ширине листа.

По умолчанию красная строка равна 0, то есть в первой строке нет отступа слева. Попробуем изменить это значение на 20:

```
WordApp.Selection.Paragraphs.FirstLineIndent:=20;
```

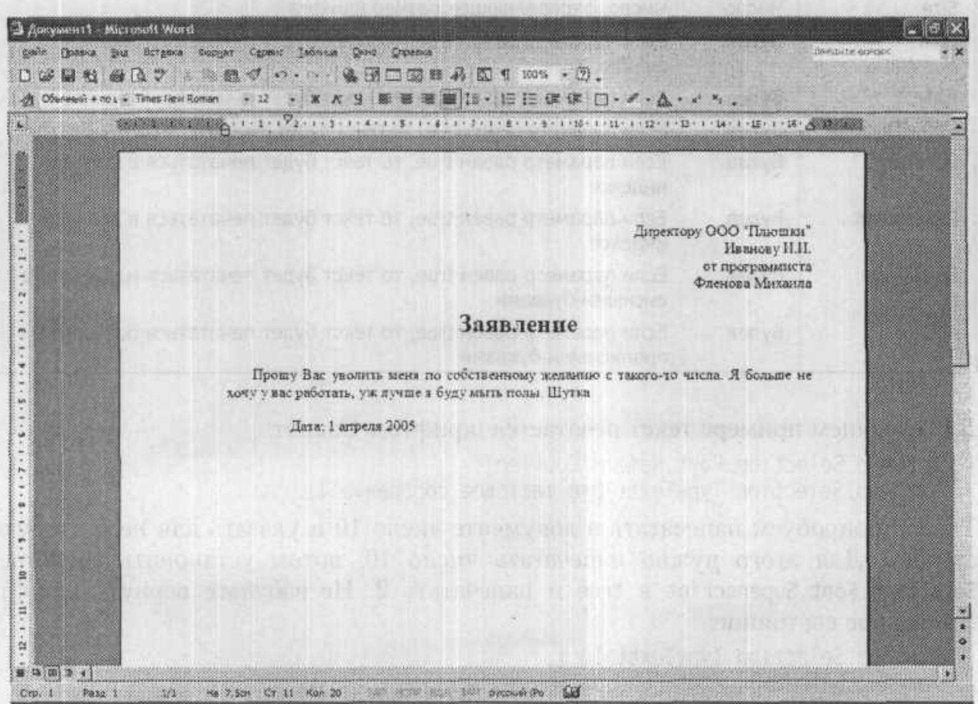


Рис. 3.16. Пример заявления

А что мы мелочимся? У нас уже достаточно информации, чтобы программно создать какой-нибудь документ. В листинге 3.28 представлен код создания заявления на увольнение, которое вы можете видеть на рис. 3.16. Немного доработав этот пример, вы сможете организовать электронный документооборот с возможностью выгрузки любого документа в MS Word.

Листинг 3.28. Программное создание документа

```
procedure TForm1.bnTestClick(Sender: TObject);
var
 WordApp, Doc: Variant;
begin
 try
  WordApp:= CreateOleObject('Word.Application');
 except
  ShowMessage('Похоже, что Word не установлен :(');
  exit;
 end;
 WordApp.Visible:=true;
 Doc:=WordApp.Documents.Add;

 // Печатаем текст
 WordApp.Selection.Font.Size:=12;
 WordApp.Selection.Paragraphs.Alignment:=2;
 WordApp.Selection.TypeText('Директору ООО "Плюшки"'+#13+#10);
 WordApp.Selection.TypeText('Иванову И.И.'+#13+#10);
 WordApp.Selection.TypeText('от программиста'+#13+#10);
 WordApp.Selection.TypeText('Фленова Михаила'+#13+#10);

 WordApp.Selection.TypeParagraph;
 WordApp.Selection.Paragraphs.Alignment:=1;
 WordApp.Selection.Font.Size:=20;
 WordApp.Selection.Font.Bold:=true;
 WordApp.Selection.TypeText('Заявление'+#13+#10);

 WordApp.Selection.TypeParagraph;
 WordApp.Selection.Paragraphs.Alignment:=3;
 WordApp.Selection.Font.Bold:=false;
 WordApp.Selection.Font.Size:=12;
 WordApp.Selection.Paragraphs.FirstLineIndent:=20;
 WordApp.Selection.TypeText('Прошу Вас уволить меня
   по собственному желанию');
 WordApp.Selection.TypeText(' с такого-то числа.
   Я больше не хочу у вас работать.');
 WordApp.Selection.TypeText(' уж лучше я буду мыть полы.
   Шутка'+#13+#10);

 WordApp.Selection.TypeParagraph;
 WordApp.Selection.Paragraphs.FirstLineIndent:=50;
 WordApp.Selection.TypeText('Дата: 1 апреля 2005');
end;
```

Работа с таблицами

Работа с таблицами не всегда интересна, но во время формирования отчетности всегда востребована. Для создания таблицы используется хитрая конструкция `WordApp.ActiveDocument.Tables.Add`, которой нужно передать пять параметров:

- ширину таблицы;
- количество строк;
- количество колонок;
- поведение таблицы по умолчанию;
- параметры заполнения.

После выполнения этого метода курсор перемещается внутрь таблицы, и вы можете начинать ввод данных в первую ячейку. Для движения между ячейками нужно использовать метод `Selection.MoveRight` с указанием в качестве первого параметра (точность) значения 12. Если вспомнить табл. 3.1, то точность со значением 12 соответствует движению между ячейками.

В движении между ячейками есть интересные нюансы. Если вы находитесь в самой последней ячейке последней строки и по идее двигаться дальше некуда, при перемещении в следующую ячейку будет создана новая строка. В следующем примере вы увидите это на практике, когда мы создадим таблицу из двух строк, а заполним три строки.

Если находиться в последней ячейки последней строки и выполнить перемещение на строку вниз (в качестве точности для метода `Move` указать 5), то вы выйдете из таблицы и окажетесь справа от нее. После этого можно перейти на строку ниже путем перемещения курсора на одну позицию вправо. Этот вариант мы также рассмотрим в следующем примере.

Итак, давайте выясним, как создать и заполнить таблицу (рис. 3.17). Я специально добавил заголовок перед таблицей и текст после таблицы, чтобы вы знали, как войти в таблицу и как выйти из нее. Код для создания такого документа показан в листинге 3.29.

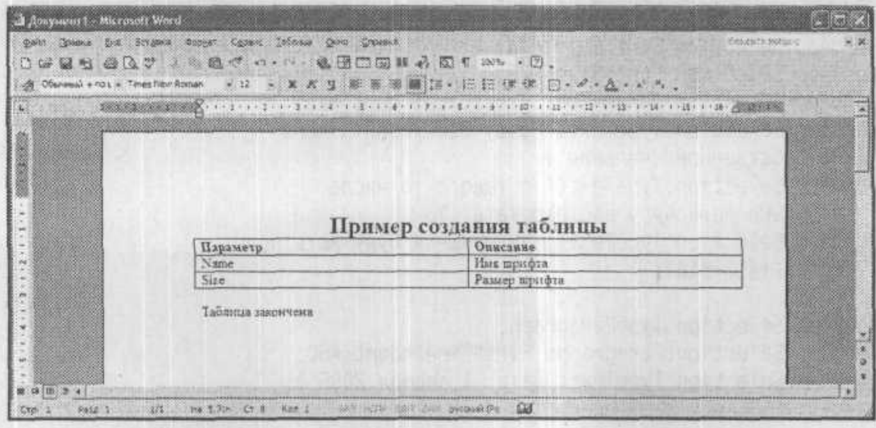


Рис. 3.17. Пример программно созданной таблицы

Листинг 3.29. Программное создание таблицы

```
procedure TForm1.bnCreateTableClick(Sender: TObject);
var
 WordApp, Doc: Variant;
begin
  try
   WordApp:= CreateOleObject('Word.Application');
  except
   ShowMessage('Похоже, что Word не установлен :(');
   exit;
  end;
  WordApp.Visible:=true;
  Doc:=WordApp.Documents.Add;

  // Печатаем заголовок
  WordApp.Selection.TypeParagraph;
  WordApp.Selection.Paragraphs.Alignment:=1;
  WordApp.Selection.Font.Size:=20;
  WordApp.Selection.Font.Bold:=true;
  WordApp.Selection.TypeText('Пример создания таблицы'+#13+#10);

  // Изменяем стиль текущего абзаца,
  // потому что он будет в таблице
  WordApp.Selection.Font.Size:=12;
  WordApp.Selection.Paragraphs.Alignment:=3;
  WordApp.Selection.Font.Bold:=false;

  // Создание таблицы
  WordApp.ActiveDocument.Tables.Add(Range:=WordApp.Selection.Range,
      NumRows:=2, NumColumns:=2,
      DefaultTableBehavior:=1,
      AutoFitBehavior:= 0);

  // Заполнение таблицы
  WordApp.Selection.Font.Bold:=true;
  WordApp.Selection.TypeText('Параметр');
  WordApp.Selection.MoveRight(12,1);
  WordApp.Selection.Font.Bold:=true;
  WordApp.Selection.TypeText('Описание');
  WordApp.Selection.MoveRight(12,1);
  WordApp.Selection.TypeText('Name');
  WordApp.Selection.MoveRight(12,1);
  WordApp.Selection.TypeText('Имя шрифта');
  WordApp.Selection.MoveRight(12,1);
  WordApp.Selection.TypeText('Size');
  WordApp.Selection.MoveRight(12,1);
  WordApp.Selection.TypeText('Размер шрифта');

  // Выход из таблицы
  WordApp.Selection.Move(5, 1);
  WordApp.Selection.Move(1, 1);
  WordApp.Selection.TypeParagraph;
  WordApp.Selection.TypeText('Таблица закончена'+#13+#10);
 end;
```

Заголовок таблицы сделан жирным шрифтом кеглем 20. После печати заголовка размер шрифта изменяется на 12 и «жирность» убирается. Зачем? Дело в том, что внутри таблицы для текста используются текущие параметры абзаца, а после печати заголовка текущим стал жирный шрифт размера 20, но такой текст в ячейках таблицы не нужен.

В представленном примере я указал достаточно интересные параметры создания таблицы.

- Ширина таблицы устанавливается в свойстве `WordApp.Selection.Range`. Что это за свойство? Свойство `Range` возвращает ширину текущего абзаца, значит, таблица будет создана на всю ширину.
- Количество колонок и строк я установил равным двум, а в примере мы заполняем три строки.
- Для таблицы по умолчанию установлено поведение –1.
- В качестве варианта заполнения устанавливается значение 0, что соответствует фиксированной ширине колонок.

После заполнения таблицы перемещаемся дальше. Мы создали таблицу размером 2×2, то есть с четырьмя ячейками, но при этом перемещаемся и заполняем шесть ячеек. Программа MS Word сама создает новую строку, если таблица заканчивается, а вы требуете переместиться в новую ячейку.

Для выхода из таблицы выполняются две строки:

```
WordApp.Selection.Move(5, 1);
WordApp.Selection.Move(1, 1);
```

Первая перемещает курсор на одну строку вниз, но реально курсор оказывается справа от таблицы. Вторая строка двигает курсор вправо, а так как таблица занимает всю ширину и вправо двигаться некуда (да и символов справа нет, чтобы перемещаться по ним), курсор оказывается на следующей за таблицей строке.

Расширение возможностей

Описать все возможности MS Word я не могу, так как этих возможностей очень много, поэтому мы рассмотрели только основы, которых тем не менее вполне достаточно для построения отчета. А теперь познакомимся с макросами, которые позволяют узнать все то, о чем мы не говорили.

Выполните следующие действия.

1. Выберите в меню команду Сервис ▸ Макрос ▸ Начать запись.
2. Перед вами появится окно, показанное на рис. 3.18. В этом окне можно указать имя макроса, а можно оставить значение по умолчанию.
3. Щелкните на кнопке OK, чтобы начать запись макроса.
4. Выполните вручную все те действия, которые вы хотите впоследствии реализовать программно, например попробуйте создать таблицу, показанную на рис. 3.17.
5. Выберите в меню команду Сервис ▸ Макрос ▸ Остановить запись.

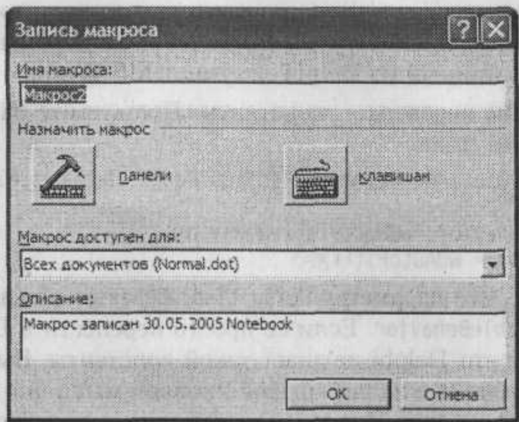


Рис. 3.18. Окно начала записи макроса

6. Теперь у нас есть макрос, который выполняет нужные действия. Для просмотра содержимого макроса выберите в меню команду **Сервис ▸ Макрос ▸ Макросы**, и перед вами появится окно, показанное на рис. 3.19.
7. Найдите имя нужного макроса и щелкните на кнопке **Изменить**. Перед вами откроется окно редактора Microsoft Visual Basic с кодом макроса.

Рис. 3.19. Выбор необходимого макроса

Посмотрите внимательно на код макроса. Вам он ничего не напоминает? Если вы, как я вас просил, на пятом шаге создали таблицу, то код вашего макроса должен напоминать код из листинга 3.29. Я просто взял содержимое макроса и вставил в Delphi, приведя синтаксис Visual Basic к правилам Delphi. На Delphi строки пишутся не в двойных кавычках, а в одинарных, а параметры заключаются в скобки. Символы подчеркивания в конце строк надо удалить (символ

подчеркивания на языке Visual Basic показывает, что код переносится на новую строку). Перед каждым параметрам я также добавил переменную WordApp, которая ссылается на созданный из Delphi документ MS Word.

Есть еще один нюанс перевода — константы. Посмотрите на код создания таблицы на языке Visual Basic:

```
ActiveDocument.Tables.Add Range:=Selection.Range, NumRows:=1,
    NumColumns:= 2,
    DefaultTableBehavior:=wdWord9TableBehavior,
    AutoFitBehavior:= wdAutoFitFixed
```

Обратите внимание, что параметру DefaultTableBehavior присваивается значение константы wdWord9TableBehavior. Если ее просто перенести в Delphi, то произойдет ошибка, потому что Delphi не знает такой константы. Выделите ее в редакторе Visual Basic, щелкните на ней правой кнопкой мыши и в контекстном меню выберите команду Quick Info. Под именем константы появится подсказка, сообщающая, что константа wdWord9TableBehavior равна 1. Именно это значение я использовал в листинге 3.29 для создания таблицы.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\Word.

Данные кубиком

Рассмотрим еще одну очень интересную задачу. Допустим, вам нужно вести учет семейных расходов. Для эффективного решения этой задачи понадобится пять таблиц.

- tFemily — таблица с личными данными всех членов семьи из двух полей — поля первичного ключа Key1 и поля с именами членов семьи vcName;
- tOutlay — таблица купленных товаров из четырех полей:
  - Key1 — первичный ключ таблицы;
  - lOutlayType — тип расходов (это поле связано с таблицей lOutlay);
  - vcName — название товара;
  - iPeriod — частота покупок товара;
- tOutlayType — название типов товаров. Эта таблица состоит из первичного ключа и название типа;
- tPeriod — периодичность покупки товара (ежедневно, единовременно и т. д.). Таблица будет состоять из двух полей, поля первичного ключа и поля PeriodName с названиями периодов;
- tPaymants — в этой таблице будем «хранить» расходы. Таблица будет состоять из следующих полей:
  - Key1 — первичный ключ;
  - dDatePayment — дата покупки товара;
  - tFemily — покупатель (это поле связано с таблицей tFemily);

- ☐ `lOutlay` — название товара (это поле связано с таблицей `lOutlay`);
- ☐ `iNumber` — количество купленного товара;
- ☐ `fCost` — цена товара.

Необходимо в программе реализовать возможность работы со справочниками типов товаров, названий товаров, членов семьи и периодичности покупок. Основной таблицей является таблица `tPaymants`, в которой хранятся расходы, и работу с этой таблицей можно реализовать в основном окне (рис. 3.20). Необходимо дать пользователю возможность добавления и удаления расходов. Сделать это достаточно просто, поэтому останавливаться на этом не будем. Самое интересное — отчеты.

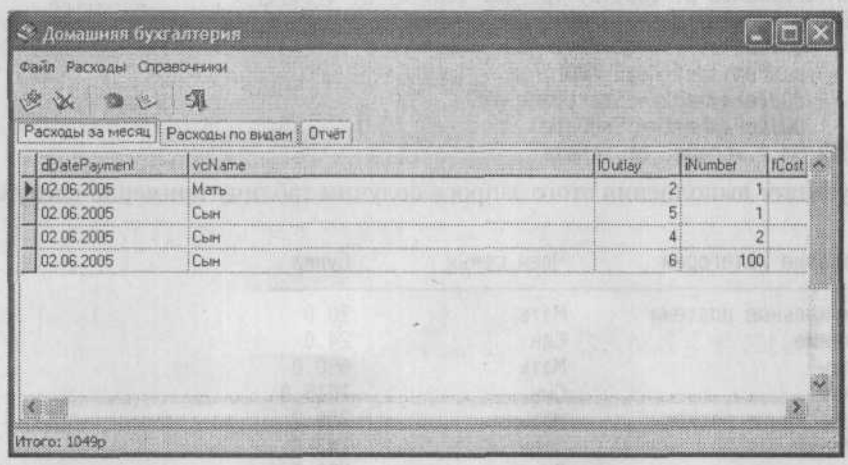


Рис. 3.20. Главное окно программы

Итак, первый отчет, который необходимо создать, — это отчет расходов по видам товаров. Очень часто нужно знать, сколько израсходовано на продукты, на хозяйственные товары, на услуги связи и т. д. Такая задача решается одним запросом, в котором самое сложное — связать три таблицы:

```
SELECT qt.vcName, SUM(fCost*iNumber) AS'Сумма'
FROM tPayments p, tOutlayType qt, tOutlay q
WHERE p.lOutlay=q.Key1 AND
      q.lOutlayType=qt.Key1 AND
      dDatePayment>=:StartD AND
      dDatePayment<=:EndD
GROUP BY qt.vcName
```

Зная язык запросов SQL, вы легко можете решить подобную задачу. Давайте разберем запрос по секциям:

- `SELECT` — здесь мы указываем тип товара и суммируем все покупки, полученные как результат умножения цены на число всех товаров данного типа;
- `FROM` — здесь мы перечисляем три таблицы (расходов, типов товаров и товаров);

- WHERE — создание связей между таблицами и ограничение расчетов только теми затратами, которые совершены в определенный период (то есть поле dDatePayment должно быть больше переменной StartD и меньше EndD);
- GROUP BY — группировка по названию типа товара.

Усложним задачу. Давайте подсчитаем сумму затрат каждого члена семьи на каждую категорию товара. Это значит, что предыдущий запрос должен быть как бы поделен на затраты каждого члена семьи. Как это сделать? В виде запроса достаточно просто сгруппировать данные не только по названию категории товара, но и по члену семьи:

```
SELECT qt.vcName, f.vcname, SUM(fCost*iNumber)
FROM tPayments p, tOutlayType qt, tOutlay q, tFemily f
WHERE p.lOutlay=q.Key1 AND
    q.lOutlayType=qt.Key1 AND
    p.tFemily=f.Key1 AND
    dDatePayment>=:StartDate AND
    dDatePayment<=:EndDate
GROUP BY f.vcname, qt.vcName
```

В результате выполнения этого запроса получим таблицу примерно следующего вида:

Название категории	Член семьи	Сумма
Комунальные платежи	Мать	10.0
Питание	Сын	24.0
Связь	Мать	650.0
Связь	Сын	1015.0
Ежемесячные расходы	Мать	300.0
Питание	Отец	143.0
Связь	Дочь	200.0
Ежемесячные расходы	Дочь	500.0

А как теперь эти данные представить таким образом, чтобы в колонках были перечислены члены семьи, а строках — типы расходов? Как после этого подсчитать сумму по строкам и по колонкам? С помощью одного запроса решить задачу нельзя, а выполнять несколько запросов неэффективно, поэтому нужно искать иное решение. Его нашли разработчики Delphi: с помощью трех компонентов DecisionCube, DecisionSource и DecisionGrid задача решается без сложного программирования, а результат представляется в удобном виде. Пример можно увидеть на рис. 3.21. Получается, что мы должны создать двухмерную таблицу из одномерного результата. При этом первое и второе поля результата должны стать размерностями (оси X и Y), а последнее — значениями таблицы.

Теперь переходим к реализации примера. Для начала поместим в модуль данных (можете использовать и главную форму) следующие компоненты.

- Компонент AdoQuery (вкладка ADO) будет выполнять рассмотренный ранее запрос на выборку данных.
- Компонент DecisionCube (вкладка DecisionCube) предназначен для форматирования результата. В его свойстве DataSet нужно указать компонент AdoQuery с нужным запросом.

■ Компонент DecisionSource (вкладка DecisionCube) представляет собой источник данных, такой же как TDataSource, только специально предназначенный для отображения данных из компонента DecisionCube. В свойстве DecisionCube нужно указать компонент форматирования запроса.

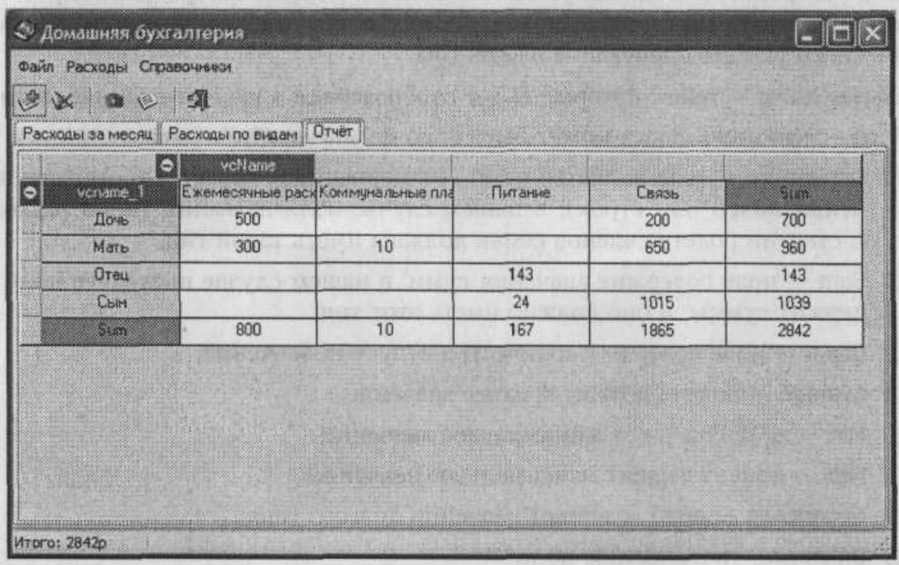


Рис. 3.21. Кубическое представление данных

Рис. 3.22. Окно настройки отображения

Попробуйте сделать запрос активным. Если в этот момент не происходит никаких ошибок, значит, программа Delphi правильно разобрала запрос и определила,

где находятся имена полей, где имена строк, а где значения. В моем случае автоматически определить это программе не удалось. Для решения проблемы нужно дважды щелкнуть на свойстве `DimensionMap` компонента `DecisionCube`, открыв окно настройки отображения (рис. 3.22).

С левой стороны окна можно увидеть все поля, которые возвращает запрос. Выбирая поле, с правой стороны можно настроить его отображение. Как минимум необходимо указать следующие параметры.

- Display Name — текст, который будет отображаться в качестве размерности.
- Type — тип поля. Здесь может быть одно из значений:
  - Dimension — поле является осью (размерностью) таблицы (содержит значения полей или строк); в нашем случае поля названий типов расходов и степени родства членов семьи должны иметь такой тип;
  - Sum — поле содержит значения сумм; в нашем случае последнее поле содержит суммы, и оно должно иметь этот тип;
  - Count — поле содержит количество непустых значений;
  - Average — поле содержит среднее значение;
  - Min — поле содержит минимальное значение;
  - Max — поле содержит максимальное значение;
  - GenericAgg — поле содержит значения другого типа;
  - Unknown — неизвестное поле.

Значение остальных параметров указывать не обязательно, поэтому мы рассмотрим только некоторые из них.

- Active Type — тип оси:
  - Active — размерность (ось) является критической и отображается всегда;
  - AsNeeded — размерность отображается только при необходимости (этот тип следует выбирать, если ось используется нечасто);
  - Inactive — ось не отображается.
- Format — формат значений. Формат задается в виде строки, как и в функции форматирования времени `FormatDateTime`.
- Base Field — имя поля базы данных, которое отображает или суммирует значение осей.

На вкладке Memory Control можно настроить параметры, которые позволят лучше управлять памятью (рис. 3.23).

- Dimensions — в этом поле можно указать максимальное количество осей. По умолчанию задано число 5. В примере мы ограничились только двумя осями, поскольку я просто не смог придумать пример, который наглядно показал бы эффективность большего количества осей.
- Summaries — максимальное количество полей сумм;
- Cells — максимальное количество ячеек. По умолчанию указано число 0, что соответствует отсутствию максимума.

- Designer Data Options — эта группа переключателей позволяет выбрать вариант представления данных во время проектирования:
  - Display Dimension Names — отображать имена осей;
  - Display Names and Values — отображать имена и значения;
  - Display Names, Values and Totals — отображать имена, значения и суммы;
  - Runtime Display Only — отображать данные только во время выполнения (во время проектирования ничего видно не будет).

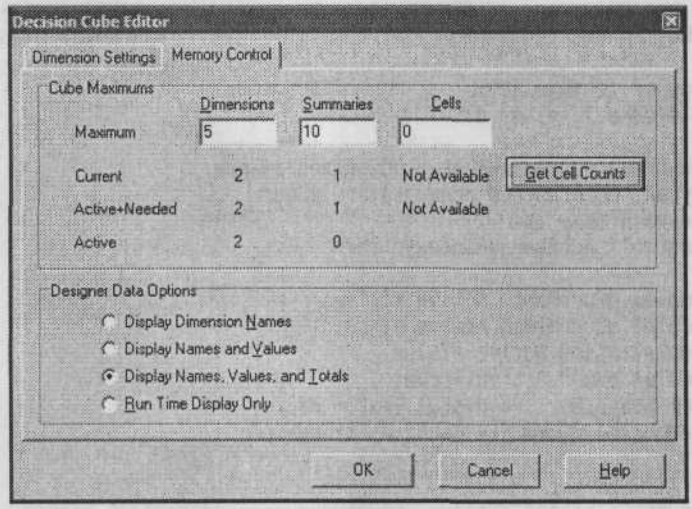


Рис. 3.23. Параметры управления памятью

Итак, мы все настроили, теперь пора все отобразить. Для этого используется компонент DecisionGrid с вкладки Decision Cube, которому достаточно в свойстве DecisionSource указать компонент DecisionSource из модуля данных, созданного нами ранее. На главной форме я установил компонент PageControl, состоящий из трех вкладок:

- Расходы за месяц — просмотр, добавление и удаление расходов;
- Расходы по видам — отчет с расходами товаров по типам;
- Отчет — это двухмерное представление данных с помощью компонента DecisionGrid.

При смене вкладки (по событию OnChange) компонента PageControl у меня выполняется код из листинга 3.30.

В зависимости от выбранной вкладки обновляется соответствующий запрос, потому что мы рассчитываем на многопользовательскую систему, где одновременно с базой данных может работать несколько пользователей. При изменении вкладки данные считываются из базы данных заново, и мы можем увидеть изменения, внесенные другими пользователями.

Из базы данных выбираются не все записи, а только записи за текущий месяц. Можете добавить возможность выбора периода, за который нужно отображать данные. Чтобы определить начало и конец текущего месяца, я декодирую текущую дату с помощью функции DecodeDate и получаю текущие месяц и год. Число нас не интересует, потому что для начальной даты в качестве числа будет использоваться 1, а в качестве конечной даты — последнее число месяца, которое можно узнать с помощью функции DaysInAMonth.

Листинг 3.30. Обработчик события OnChange компонента PageControl

```
procedure TMainForm.PageControl1Change(Sender: TObject);
var
 y,m,d:WORD;
begin
 // Декодируем текущую дату
 DecodeDate(Date(), y, m, d);

 // Если выбрана первая вкладка, обновляем данные
 // на случай, если другой пользователь внес
 // изменения в базу данных
 if PageControl1.ActivePageIndex=0 then
  begin
   DataModule1.qCurrMonth.Active:=false;
   DataModule1.qCurrMonth.Active:=true;
   DataModule1.qSumm.Active:=false;
   DataModule1.qSumm.Active:=true;
   MainForm.StatusBar1.Panels[0].Text:='Итого: '+
      DataModule1.qSumm.Fields[0].AsString+'p';
  end;

 // Если выбрана вторая вкладка, обновляем соответствующий запрос
 if PageControl1.ActivePageIndex=1 then
  begin
   with DataModule1.qQuery do
    begin
     Active:=false;
     Parameters.ParamValues['StartD']:=EncodeDate(y, m, 1);
     Parameters.ParamValues['EndD']:=EncodeDate(y, m, DaysInAMonth(y, m));
     Active:=true;
    end;
  end;

 // Если выбрана вкладка "Отчет", обновляем запрос отчета
 if PageControl1.ActivePageIndex=2 then
  begin
   with DataModule1.qReport do
    begin
     Active:=false;
     Parameters.ParamValues['StartDate']:=EncodeDate(y, m, 1);
     Parameters.ParamValues['EndDate']:=EncodeDate(y,m,DaysInAMonth(y, m));
     Active:=true;
    end;
  end;
end;
```

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\HomeBuh.

Давайте усложним задачу и создадим трехмерную таблицу. Для этого в запрос нужно добавить еще одно поле, например поле названия товара. Таким образом, у нас получатся три оси: категория товара, название товара, степень родства членов семьи. Запрос будет выглядеть следующим образом:

```
SELECT qt.vcName, f.vcname, q.vcName, SUM(fCost*iNumber)
FROM tPayments p, tOutlayType qt, tOutlay q, tFemily f
WHERE p.lOutlay=q.Key1 AND
    q.lOutlayType=qt.Key1 AND
    dDatePayment>=:StartDate AND
    dDatePayment<=:EndDate AND
    p.tFemily=f.Key1
GROUP BY f.vcname, qt.vcName,q.vcName
```

Теперь выделяем компонент DecisionCube и дважды щелкаем на свойстве DimensionMap. Новому полю нужно установить свойство Dimension. Запустите программу, и в результате у нас получится уже трехмерная таблица (рис. 3.24).

Домашняя бухгалтерия

Файл Расходы Справочники

Расходы за месяц | Расходы по видам | Отчёт

		Тип товара			
Член семьи	Товар	Ежемесячные расх	Питание	Связь	Sum
Дочь	Интернет			1800	1800
	Проезд	500			500
	Сотовый телефон			200	200
	Хлеб		8		8
	Sum	500	8	2000	2508
Sum		500	8	2000	2508

Итого: 2850р.

Рис. 3.24. Трехмерная таблица

Обратите внимание, что в результирующей таблице остался только один член семьи, а остальные куда-то пропали. Почему? Проблема в запросе, а именно в последовательности, в которой перечислены поля: тип товара, степень родства членов семьи, товар. Здесь имеет место нарушение связи. Тип товара можно связать со степенью родства членов семьи только через сам товар. Попробуем изменить последовательность следующим образом: степень родства членов семьи, тип товара, товар:

```
SELECT f.vcname, qt.vcName, q.vcName, SUM(fCost*iNumber)
FROM tPayments p, tOutlayType qt, tOutlay q, tFemily f
WHERE p.lOutlay=q.Key1 AND q.lOutlayType=qt.Key1
   AND dDatePayment>=:StartDate and dDatePayment<=:EndDate
   AND p.tFemily=f.Key1
GROUP BY f.vcname, qt.vcName, q.vcName
```

После этого получим более изящную таблицу (рис. 3.25). Как видите, результат зависит от последовательности, в которой перечислены поля. Учитывайте это, когда создаете трехмерную таблицу.

vcName_1	vcName_2	Дочь	Мать	Отец	Сын	Sum
Ежемесячные расх	Проезд	500	300			800
	Sum	500	300			800
Коммунальные пла	Вода		10			10
	Sum		10			10
Питание	Молоко			143	24	167
	Хлеб	8				8
	Sum	8		143	24	175
Связь	Интернет	1800			1000	2800
	Сотовый телефон	200	650		15	865
	Sum	2000	650		1015	3665
Sum		2508	960	143	1039	4650

Рис. 3.25. Трехмерная таблица с измененной последовательностью полей

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch03\HomeBuh2.

Я надеюсь, что описанные в этой главе алгоритмы помогут вам при создании собственных приложений. Благодаря им я очень быстро создаю приложения, работающие с базами данных. Главное — везде использовать шаблонные документы, которые впоследствии позволят с минимальными затратами наращивать функциональность программы.

Необходимо учитывать, что код некоторых процедур не оптимален. Это сделано по двум причинам:

- чтобы код был более понятным и легче воспринимался;
- чтобы добиться максимальной универсальности кода.

Вы можете оптимизировать процедуры, но этим вы добьетесь лишь минимального прироста в производительности. Хотя код и не оптимален, откровенно слабых мест в нем нет. Чтобы добиться реального повышения скорости работы, которое будет заметно даже на глаз, необходимо отказаться от универсальности и адаптировать каждое действие к конкретной ситуации, но тогда вы потеряете в простоте поддержки и расширения своих проектов.

Обратите внимание, что я не использовал наследование форм. Не делайте этого и вы. Это самое глупое решение, которое приносит только проблемы. Мне при-

ходилось дорабатывать чужие программы после того, как их создатели уволились. Сопровождая такой код, сразу понимаешь, почему программисты увольняются. Они просто не хотят сопровождать свои творения. Старайтесь писать так, чтобы вам самим удобно было сопровождать свой код, ведь именно поддержка уже готовых программ отнимает у любого программиста до 90 % рабочего времени.

Я прибегаю к наследованию форм только в крайних случаях, причем не снабжаю формы предков визуальным интерфейсом. Именно визуальный интерфейс чаще всего приводит к проблемам, потому что его наследование (особенно многоуровневое, когда имеются несколько предков форм) — достаточно сложное занятие.

Не злоупотребляйте визуальным редактированием форм. Лучше написать лишнюю строку кода, но упростить себе последующую жизнь, чем тыкать мышью в сотню похожих форм. Поверьте, это утомляет. Особенно, когда приходится вносить изменения. В программе управления клиентами для изменения функциональности справочников достаточно внести изменения только в одну форму, и все справочники заработают по-новому.

Например, если заменить в шаблоне формы для работы справочника стандартную сетку сеткой `CyDDBGrid` и добавить на форму кнопку для выгрузки справочников в Excel, то выгружать все справочники в Excel можно будет двумя щелчками.

Подобное построение кода позволяет с легкостью наращивать возможности программы. Я затратил на написание этого шаблона целую неделю, а потом еще за неделю написал на его основе большую программу управления базами данных. Мои шаблоны можно расширять; возможно, когда-нибудь вы создадите из них что-нибудь наподобие 1С. Хотя на написание кода для решения бухгалтерских задач нужно очень много времени, но что-то в стиле складской программы с удобными и мощными возможностями может быть создано в максимально короткие сроки.

В одной главе небольшой книги мы не могли рассмотреть абсолютно все методы написания универсального кода. Мы познакомились только с основами, а в каждом конкретном случае нужно действовать в зависимости от ситуации. Чтобы создавать универсальные решения, при написании собственных проектов следуйте следующим правилам.

- Не привязывайтесь к определенным именам. Если можно абстрагироваться от объектов, необходимо делать это. Например, в своих проектах я никогда не обращаюсь к компонентам `TADOTable` и `TADOQuery` напрямую. Вместо этого я обращаюсь к ним через свойство `DataSource` компонента сетки; таким образом, к сетке может быть привязана любая таблица или запрос, а форма будет работать корректно.
- Используйте такой механизм именования компонентов, который заставляет их работать схожим образом. В разделе «Создание сервера» я таким образом сделал все компоненты `TADOTable` активными при старте сервера. Если какая-то

таблица должна быть активной, то ее имя должно начинаться с букв tb. Если в активности нет необходимости, то достаточно изменить первые буквы, и на старте такой компонент активироваться уже не будет.

- Разбивайте код на отдельные процедуры, которые позволят делать его универсальным, как мы это делали с механизмами сортировки или поиска. Такие процедуры впоследствии достаточно просто адаптировать для других проектов.

Помните, что универсальность позволит сэкономить время на сопровождение уже существующего проекта и сократить время разработки будущих проектов. Один раз написав модуль с функцией для универсальной сортировки, достаточно только подключить этот модуль к новому проекту и использовать.

Глава 4

Алгоритмы

В этой главе нам предстоит познакомиться с различными технологиями программирования и интересными алгоритмами. Алгоритм — это основа программы. От правильно выбранного алгоритма зависит ее будущее. Вообще, это достаточно сложная тема, ведь качество алгоритма во многом определяется знанием API-функций, которые предоставляет нам ОС Windows и среда разработки Delphi. Однако этих функций очень много, и даже простой справочник с короткими описаниями окажется объемнее, чем труд великого русского писателя «Война и мир».

Почему алгоритмы так важны? Я очень часто пишу в своих работах об оптимизации, но как бы вы ни оптимизировали код программы, в основе которого — медленный алгоритм, этот код всегда будет работать медленнее, чем неоптимизированный код, но разработанный на базе быстрого алгоритма.

Зачем нужно оптимизировать программы, которые на современных процессорах и так работают быстро? Я понимаю, что Pentium 4 может просчитать практически что угодно достаточно быстро, но только если все процессорное время будет предоставлено исключительно вашей программе. А если одновременно копировать файл, проверять систему на вирусы и кодировать аудио и/или видео, то неоптимизированный код и плохой алгоритм проявятся сразу. Именно поэтому даже самую простую программу нужно писать тщательно и аккуратно. Пользователи оценят ваш труд, когда даже при максимальной нагрузке системы с программой можно будет работать.

Если вы хотите получить больше информации по теории оптимизации, рекомендую обратиться к моей книге «Delphi в шутку и всерьез: что умеют хакеры», вышедшей в издательстве «Питер» в 2004 г. А в данной главе мы затронем практические аспекты *оптимизации*, познакомившись с процессом разработки и оптимизации реального алгоритма. Результаты оптимизации особенно заметны в графических программах, код которых можно оптимизировать практически бесконечно.

Помимо алгоритмов в данной главе мы познакомимся с некоторыми технологиями, призванными помочь вам в будущем при создании собственных приложений. Как всегда, мы напишем множество интересных примеров, код которых можно будет легко адаптировать для решения своих задач. Я постарался подготовить для вас примеры, в которых представлены решения не только основной задачи, но и сопутствующих ей проблем.

RTTI

Как получить информацию о человеке? Первое, что приходит на ум, — расспросить его обо всем, что вас интересует. Нужно задать множество вопросов, каждый из которых будет определять один из параметров. Такой вариант называют анкетированием.

Недостаток анкеты заключается в необходимости задавать множество вопросов. Намного проще просто взять паспорт человека и сразу выяснить все, что мы хотим знать.

Как получить информацию о компоненте Delphi? Если мы четко знаем его тип, то это не так сложно, ведь всю информацию можно взять из файла помощи. Но если компонентов много и их тип заранее не известен? В этом случае в коде придется писать множество конструкций (вопросов) `if..then`, но тогда код получится слишком трудноуправляемым.

В современных языках программирования появилась очень мощная и удобная технология RTTI (Run Time Type Information — информация о типах во время выполнения). В Delphi с помощью этой технологии можно определить намного больше, чем просто тип. Например, с помощью RTTI-функций можно определить свойства, методы и события объекта.

Постановка задачи

Давайте создадим что-нибудь интересное в стиле дизайнера форм с объектным инспектором, наподобие дизайнера форм Delphi. Где может пригодиться такой пример? Допустим, вам требуется создать программу моделирования тех или иных процессов. Это можно сделать, расставив на форме компоненты, имитирующие какой-либо процесс, и отобразив связи между ними. Компоненты на форме можно реализовать в виде потомков класса от `TControl` (они готовы к работе на форме). Достаточно наделить их нужными свойствами и методами и с помощью RTTI заставить все компоненты работать нужным образом.

В нашем примере будет создаваться что-то похожее на структуру базы данных. Главное окно программы будет являться визуальным редактором, в котором можно расставлять на форме графические изображения таблиц. При выделении изображения таблицы в объектном инспекторе появятся его свойства, включая расположение компонента, его имя и другие параметры.

О том, как сформировать главное окно (рис. 4.1), я рассказывать не буду. Надеюсь, вы сможете воссоздать его по рисунку. А основной упор сделаем на алгоритмы.

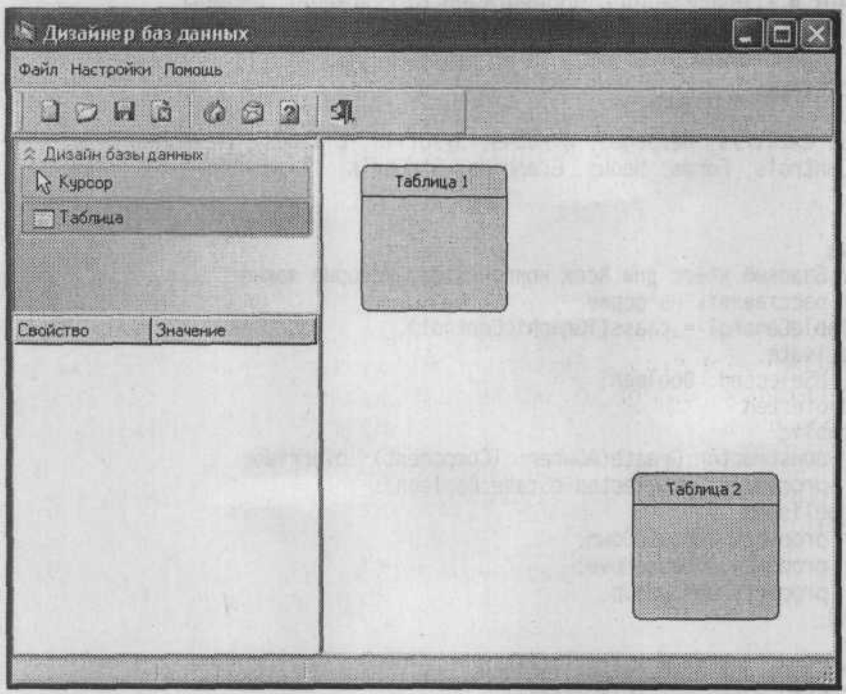


Рис. 4.1. Главное окно будущей программы

Наделим дизайнер богатыми возможностями, которые позволят нам познакомиться с интересными алгоритмами и узнать что-то новое. А нового будет предостаточно. Библиотека для работы с типами документирована очень плохо, точнее сказать, в файле помощи, который поставляется с Delphi, практически ничего нет. Мне приходилось все изучать по исходным кодам. Именно там я нашел необходимые функции и по исходным кодам выяснил, как с ними работать.

Реализация дизайнера

Итак, на форме будем расставлять компоненты, которые должны изображать таблицы и хранить их свойства. Из стандартных компонентов ничего не подходит, поэтому придется создать собственный компонент, который будет потомком от класса `TGraphicControl`. Почему выбран именно этот предок? Его можно устанавливать на форму, и на нем можно рисовать, а это позволяет представлять свойства таблиц графически.

Хотя в нашей программе на форму будут устанавливаться только таблицы, код мы сделаем универсальным, и вы легко его сможете расширить (листинг 4.1). Сам компонент будет просто храниться в модуле, то есть никаких процедур для регистрации компонента в Delphi мы создавать не будем, поэтому его не удастся установить в дизайнере форм Delphi. Да нам это и не нужно, поскольку такие компоненты будут создаваться только во время выполнения программы.

Листинг 4.1. Реализация компонента для отображения таблицы

```
unit EntityComponentUnit;

interface

uses ExtCtrls, Messages, Windows, SysUtils, Classes,
  Controls, Forms, Menus, Graphics, StdCtrls;


type
 // Базовый класс для всех компонентов, которые можно
 // расставлять на форме
 TTableControl = class(TGraphicControl)
  private
    bSelected: Boolean;
  protected
  public
    constructor Create(AOwner: TComponent); override;
    procedure SetSelected(bState:Boolean);
  published
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
 end;

 // Класс для отображения таблицы
 TEntity = class(TTableControl)
  private
    FTableName: String;
    procedure SetTableName(const Value: String);
  protected
    procedure Paint; override;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property TableName:String read FTableName write SetTableName;
 end;

implementation

// Конструктор базового класса
constructor TTableControl.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  bSelected:=false;
end;

// Метод SetSelected позволяет назначить компоненту состояние "выделен"
procedure TTableControl.SetSelected(bState: Boolean);
begin
 bSelected:=bState;
 Repaint;
end;
```

```
// Конструктор компонента таблицы
constructor TEntity.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  // Задаем размеры по умолчанию
  Width := 105;
  Height := 105;
end;

// Метод прорисовки компонента таблицы
procedure TEntity.Paint;
begin
 if bSelected then
  Canvas.Pen.Color:=clRed
 else
  Canvas.Pen.Color:=clBlue;
 Canvas.Brush.Color:=clBtnFace;
 Canvas.RoundRect(0, 0, Width, Height, 10, 10);
 Canvas.TextOut((Width-Canvas.TextWidth(FTableName)) div 2, 4, FTableName);
 Canvas.MoveTo(1, Canvas.TextHeight(FTableName)+8);
 Canvas.LineTo(Width, Canvas.TextHeight(FTableName)+8);
end;

// Установить имя таблицы
procedure TEntity.SetTableName(const Value: String);
begin
  FTableName := Value;
end;

end.
```

В этом модуле объявлено два объекта.

- TTableControl — потомок от TGraphicControl. Это базовый объект для всех компонентов, которые должны будут устанавливаться на форму. Здесь необходимо реализовывать общие свойства и методы. На данный момент я реализовал только возможность назначения компоненту свойства «выделен».
- Tentity — потомок от TTableControl. Этот компонент будет отображать таблицу.

При создании табличного компонента я пока реализовал только один метод Paint, который красиво рисует таблицу и ее свойства. А среди свойств пока есть только имя — TableName. Напоминаю, что свойства должны описываться в разделе published:

```
property имя : тип
```

В описание могут быть добавлены параметры чтения и записи свойства, но только если это необходимо. В нашем случае объявлено свойство, в котором хранится имя таблицы, и требуется обеспечить возможность его чтения и записи.

Теперь реализуем возможность добавления таблиц на форму во время выполнения программы. Для этого на главной форме установим кнопку. Если кнопка нажата, то по щелчку в рабочей области окна в месте щелчка будет создаваться табличный компонент. Соответствующий код можно увидеть в листинге 4.2.

Листинг 4.2. Создание табличного компонента

```
procedure TVRDatabaseModelerForm.sbModelAreaMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
 Entity: TEntity;
 sTableName:String;
begin
 //Create Table
 if acTableTool.Checked then
  begin
   sTableName:='';
   if not InputQuery('Имя таблицы', 'Введите имя таблицы', sTableName) then
    exit;

   Entity := TEntity.Create(nil);
   Entity.Left:=X- sbModelArea.HorzScrollBar.Position;
   Entity.Top :=Y- sbModelArea.VertScrollBar.Position;
   Entity.Width:=100;
   Entity.Height:=100;
   Entity.TableName:=sTableName;
   Entity.OnMouseDown:=NetComponentBoxMouseDown;
   Entity.OnMouseMove:=NetComponentBoxMouseMove;
   Entity.OnMouseUp:=NetComponentBoxMouseUp;
   sbModelArea.InsertControl(Entity);
  end;
end;
```

При щелчке на кнопке acTableTool выдается запрос на ввод имени новой таблицы. Если пользователь вводит имя, создается экземпляр компонента TEntity и устанавливаются его свойства, включая имя, положение и размеры таблицы, а также задаются обработчики событий OnMouseDown, OnMouseMove и OnMouseUp. В этих обработчиках мы реализуем возможность определения свойств компонента с отображением их в объектном инспекторе во время выполнения программы, а также возможность перетаскивания компонента на форме.

Для создания обработчиков давайте их сначала объявим в разделе private главной формы следующим образом:

```
procedure NetComponentBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
procedure NetComponentBoxMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
procedure NetComponentBoxMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

Опишите эти процедуры и нажмите клавиши Ctrl+Shift+C, чтобы программа Delphi создала заготовки для процедур. Сам код процедур можно увидеть в листинге 4.3.

Листинг 4.3. Обработчики событий OnMouseDown, OnMouseMove, OnMouseUp

```
// Событие OnMouseDown возникает при нажатии кнопки мыши
procedure TVRDatabaseModelerForm.NetComponentBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  UpdateProperties(Sender);
```

```
    // Активация кнопок для работы с выделенным компонентом
    EnableControlsButtons(true);

    // Подготовка к перетаскиванию
    bDragging:=true;
    iStartDraggingX:=X;
    iStartDraggingY:=Y;
    SetCaptureControl(TControl(Sender));
    TTableControl(Sender).SetSelected(true);
  end;

// Событие OnMouseMove возникает при перемещении мыши
procedure TVRDatabaseModelerForm.NetComponentBoxMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  if bDragging then
   begin
    TControl(Sender).Left:=TControl(Sender).Left+X-iStartDraggingX;
    TControl(Sender).Top:=TControl(Sender).Top+Y-iStartDraggingY;
   end;
end;

// Событие OnMouseUp возникает при отпускании кнопки мыши
procedure TVRDatabaseModelerForm.NetComponentBoxMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
Begin
  // Перетаскивание закончено
  bDragging:=false;

  // Обнуляем компонент, который перехватывает события мыши,
  // поскольку это больше не требуется
  SetCaptureControl(nil);

  // Убираем выделение с компонента
  TTableControl(Sender).SetSelected(false);

  // Обновляем свойства перемещенного компонента
  UpdateProperties(Sender);
end;
```

Когда пользователь нажимает кнопку мыши на компоненте, вызывается обработчик события OnMouseDown. В нем выполняется много интересного, поэтому остановимся на этом подробно. Первым делом вызываем метод UpdateProperties. Этого метода пока у нас нет, но он должен обновлять объектный инспектор, заполняя его свойствами компонента, который передан в качестве параметра. Мы в качестве параметра передаем переменную Sender — в ней находится указатель на компонент, на котором была нажата кнопка мыши.

В следующей строке вызывается метод EnableControlsButtons. В нашей программе это будет просто заглушка — если вы захотите расширить пример, напишите вместо заглушки требуемый код. В этом методе нужно будет активировать кнопки, которые позволяют работать с компонентами, например, кнопки просмотра свойств таблицы. Если ни один из компонентов не выделен, то кнопки должны

оставаться недоступными и вы должны будете вызвать метод EnableControlsButtons, передав ему в качестве параметра значение false.

Далее делаем логическую переменную bDragging активной. Это значит, что кнопка мыши нажата, и при движении мыши должно происходить перетаскивание выделенного объекта.

В следующих двух строках кода запоминается позиция, в которой была нажата кнопка мыши. Исходя из этих значений, мы будем перетаскивать компонент:

```
iStartDraggingX:=X;
iStartDraggingY:=Y;
```

Так как компонент выделен и его можно перетаскивать, необходимо сделать так, чтобы все движения мыши передавались этому компоненту, иначе перемещение компонента может происходить рывками или он может потерять фокус ввода. Для этого вызываем метод SetCaptureControl и передаем ему в качестве параметра указатель на компонент, на котором была нажата кнопка мыши. После этого все события от мыши обязательно будут передаваться этому компоненту.

В последней строке устанавливаем свойство Selected выделенного компонента в true с помощью метода SetSelected. Вот здесь и проявляется преимущество того, что мы создали два объекта TTableControl и TEntity. На форме могут расставляться не только компоненты TEntity, но и другие компоненты, производные от TTableControl. И им необходим только один обработчик событий OnMouseClick, который может работать с любым типом.

Так как все компоненты, расставляемые на форме, происходят от TTableControl, у них у всех есть метод SetSelected, и мы можем к нему обратиться через предка TTableControl.

При движении мышью (событие OnMouseMove) мы изменяем текущую позицию компонента, если свойство bDragging установлено в true.

Когда пользователь отпускает кнопку мыши, генерируется событие OnMouseUp, в котором свойство bDragging устанавливается в false, снимается выделение с компонента и обновляется информация в объектном инспекторе. Так как положение компонента могло измениться, могли измениться и его свойства.

Определение свойств компонента

Теперь переходим к методу UpdateProperties, который обновляет свойства компонента (листинг 4.4). Здесь реализовано не только все самое сложное, но и самое интересное.

Листинг 4.4. Обновление свойств компонента

```
procedure TVRDatabaseModelerForm.UpdateProperties(Sender: TObject);
var
 Props: PPropList;
 TypeData: PTypeData;
 i: Integer;
begin
 // Очистка текущих свойств
 vleComponent.Strings.Clear;
```

```
  // Определение количества свойств
  TypeData := GetTypeData(Sender.ClassInfo);
  if (TypeData = nil) or (TypeData^.PropCount = 0) then
   exit;

  // Выделяем память для хранения списка свойств
  GetMem(Props, TypeData^.PropCount * sizeof(Pointer));
  try
   // Получаем список свойств
   GetPropInfos(Sender.ClassInfo, Props);
   for i := 0 to TypeData^.PropCount-1 do
    begin
     with Props^[i]^ do
      begin
       // Строковое свойство
       if (PropType^^.Kind = tkLString) then
         vleComponent.Strings.Add(Name+'='+
          GetPropValue(Sender, Name));

       // Целочисленное свойство
       if (PropType^^.Kind = tkInteger) then
         vleComponent.Strings.Add(Name+'='+
          IntToStr(GetPropValue(Sender, Name)));
      end;
    end;
  finally
   // Очистка памяти
   Freemem(Props);
  end;
 end;
```

В первой строке мы очищаем содержимое компонента vleComponent. Это компонент типа TValueListEditor, в котором отображаются свойства. Очистка необходима, чтобы избавиться от свойств, которые могли принадлежать предыдущему компоненту.

Теперь начнем рассматривать RTTI-функции. Для их использования необходимо подключить модуль TypInfo.

Первая функция, которую мы вызываем — GetTypeData — определяет объем данных, необходимый для хранения всех свойств указанного в качестве параметра класса. Точнее сказать, в качестве параметра ей необходимо передать свойство ClassInfo компонента.

Результат выполнения сохраняется в переменной TypeData типа PTypeData. Если результат равен нулю, то есть количество полученных свойств равно нулю (TypeData^.PropCount), то можно выходить из функции, даже не пытаясь определить что-то еще.

Далее необходимо выделить память для переменной Props типа PPropList. Эта переменная предназначена для хранения полученного списка свойств, поэтому для нее должно быть выделено достаточно памяти. Именно для выделения памяти мы определяли количество свойств.

Итак, в переменную Props будут записываться указатели на свойства. Получается, что для определения необходимого объема памяти, нужно умножить количество свойств (TypeData^.PropCount) на размер переменной типа указателя (sizeof(Pointer)). Именно такой размер выделяется с помощью функции GetMem.

Далее можно непосредственно получить список свойств. Для этого используется функция GetPropInfos, которой необходимо передать два параметра:

- свойство ClassInfo, параметры которого нужно узнать;
- переменную типа PPropList, куда будет записан результат.

Если во втором параметре будет указана переменная с недостаточным количеством памяти или неверным указателем, то может произойти сбой программы, поэтому необходимо быть аккуратным и не забывать о выделении памяти.

После этого запускаем цикл перебора всех свойств, в котором будем определять их имена и значения:

```
for i := 0 to TypeData^.PropCount-1 do
 begin
  // Определение имени и значения текущего свойства
 end;
```

Свойства находятся в списке по адресу, на который указывает переменная Props. Раз это указатель, то его нужно разыменовывать (Props^). Чтобы получить определенный элемент, его индекс нужно указать в квадратных скобках (Props^[i]). Но это снова будет указатель на свойства i-го элемента. В результате получается немного страшная конструкция Props^[i]^, указывающая на параметры i-го свойства. Среди параметров свойства можно найти следующие:

- PropType — указатель на структуру, определяющую тип свойства. Структура обладает двумя полями:
  - Kind — разновидность свойства;
  - Name — имя свойства, которое отображается в объектном инспекторе.
- GetProc — указатель на процедуру, которая используется для чтения параметра.
- SetProc — указатель на процедуру, с помощью которой изменяется значение свойства.
- CtoredProc — указатель на функцию stored.
- Index — в некоторых свойствах может устанавливаться параметр index, который можно прочитать в поле Index. Если его нет, вы увидите число –2147483648.
- Default — значение по умолчанию. Если его нет, вы увидите число –2147483648.
- NameIndex — индекс свойства в списке.
- Name — имя свойства.

Очень интересным является поле Kind, в котором указывается тип свойства. В таблице 4.1 перечислены основные типы свойств, с которыми вы можете встретиться.

В примере будем выводить в объектном инспекторе только целочисленные и строковые свойства, то есть свойства, в которых поле Kind равно tkInteger или tkLString

соответственно. Например, чтобы сравнить свойство `Kind` на соответствие целочисленному типу, необходимо выполнить следующую проверку:

```
if (Props^[i]^.PropType^^.Kind = tkLString) then
```

Таблица 4.1. Типы свойств

Тип (Kind)	Описание
tkUnknown	Тип неизвестен. Чаще всего это характерно для пользовательских типов
tkInteger	Целое число. Примерами свойств такого типа являются свойства Tag, Top, Left и т. д.
tkChar	Один символ. Свойства такого типа используются очень редко
tkEnumeration	Перечисление. Яркий пример — HelpType или Cursor, то есть свойства с раскрывающимся списком значений
tkFloat	Число с плавающей точкой. Используется редко, потому что в свойствах чаще нужны целые числа
tkString	Короткая строка. В настоящее время практически не используется
tkLString	Длинная строка. Этот тип стал заменой типа tkString
tkClass	Объект. Яркий пример такого свойства — Font
tkSet	Набор данных, например Anchors или Options
tkMethod	Событие, на которое может реагировать компонент

Чтобы определить значение свойства, требуется выполнить функцию `GetPropValue`, передав ей два параметра:

- класс, свойство которого нужно определить;
- имя свойства.

Функция возвращает значение типа `Variant`. Для целочисленных свойств оно будет восприниматься как число, для строковых — как строка и т. д. Это значит, что для вывода в объектном инспекторе значения его для целочисленных свойств надо преобразовывать в строку.

Запустите приложение и установите на форму пару таблиц. Убедитесь, что при выделении таблицы в нашем самодельном объектном инспекторе правильно отображаются все целочисленные и строковые свойства. Обратите внимание, что свойство `TableName`, которое мы сами создали в объекте `TEntity`, также присутствует в списке (рис. 4.2).

Свойство	Значение
Name	
Tag	0
Left	117
Top	70
Width	100
Height	100
Cursor	0
Hint	
HelpKeyword	
HelpContext	0
TableName	Таблица

Рис. 4.2. Самодельный объектный инспектор

Как видите, с помощью RTTI-функций можно легко написать собственный объектный инспектор, напоминающий объектный инспектор Delphi. Среда разработки Delphi сама использует эти функции.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch04\1-Database Modeler.

Сохранение свойств компонента

Теперь давайте научим наш объектный инспектор изменять значения свойств. Точнее сказать, он это уже умеет, но пока не может сохранять измененные значения в качестве свойств компонента. Вряд ли имеет смысл делать объектный инспектор, который не умеет обновлять свойства компонента. Давайте заполним этот пробел.

Для начала необходимо подправить код обработчиков событий OnMouseDown и OnMouseUp компонента. При нажатии кнопки мыши он выделяется, а при отпускании кнопки выделение с него снимается. Снимать выделение больше не будем, поэтому убираем следующую строку из обработчика события OnMouseUp (в нашем случае это процедура NetComponentBoxMouseUp):

```
TTableControl(Sender).SetSelected(false):
```

А вот при нажатии пользователем кнопки мыши на компоненте (по событию OnMouseDown) мы должны сначала снять выделение со всех компонентов, а затем установить свойство Selected только для того компонента, на котором была нажата кнопка мыши. Для этого добавим следующий код в конец процедуры NetComponentBoxMouseDown:

```
for i:=0 to sbModelArea.ControlCount-1 do
  TTableControl(sbModelArea.Controls[i]).SetSelected(false);
TTableControl(Sender).SetSelected(true):
```

Теперь, когда пользователь нажимает кнопку мыши на компоненте, компонент выделяется и остается выделенным, пока не будет выделен другой компонент. Так сразу же видно, свойства какого компонента отображаются в объектном инспекторе и свойства какого компонента мы должны изменять.

Для компонента объектного инспектора создаем обработчик события OnSelEditText. Он вызывается каждый раз, когда нужно ввести в ячейку новое значение. К сожалению, он вызывается слишком часто: не только когда пользователь изменяет данные, но даже тогда, когда ячейки заполняются значениями свойств программно, — что не так страшно, ведь лучше лишний раз записать данные, чем не записать и потерять. Этого пользователь нам не простит.

Код обработчика OnSelEditText представлен в листинге 4.5.

Листинг 4.5. Сохранение значения свойства

```
procedure TVRDatabaseModelerForm.vleComponentSetEditText(Sender: TObject:
  ACol, ARow: Integer: const Value: string):
var
 PropInfo: PPropInfo:
```

```
  i:Integer;
 begin
  for i:=0 to sbModelArea.ControlCount-1 do
   if TTableControl(sbModelArea.Controls[i]).GetSelected then
   begin
     PropInfo := GetPropInfo(sbModelArea.Controls[i].ClassInfo,
             vleComponent.Cells[0, ARow]);
     if PropInfo=nil then exit;
     case PropInfo^.PropType^.Kind of
      tkInteger:
        SetOrdProp(sbModelArea.Controls[i], PropInfo,
         StrToIntDef(Value, 0));
      tkLString:
        SetStrProp(sbModelArea.Controls[i], PropInfo,
          Value);
     end;
     TTableControl(sbModelArea.Controls[i]).Repaint;
   end;
 end;
```

Если изменен какой-либо текст, то будет вызвана эта процедура. Ей передаются следующие параметры:

- Sender — указатель на объект, который сгенерировал событие. В данном случае это всегда будет наш компонент объектного инспектора, потому что обработчик обрабатывает только его событие OnSelEditText.
- ACol и ARow — колонка и строка ячейки, в которой нужно изменить текст;
- Value — новое значение ячейки.

Параметры очень важны, потому что именно через них передается новое значение, а в ячейке в момент выполнения этого обработчика событий будет находиться старое значение.

Итак, в процедуре запускается цикл, в котором мы ищем выделенный компонент. Если компонент выделен, то определяем параметры свойства, значение которого нужно изменить. Для этого вызывается функция GetPropInfo, которой передаются два параметра:

- Указатель на компонент, параметры свойства которого нужно определить. В нашем случае — это компонент выделенной таблицы.
- Имя свойства. Как его определить? В процедуру нам передается строка, значение в которой нужно изменить. В этой строке первая колонка — название свойства, а вторая — значение. Мы передаем процедуре GetPropInfo содержимое первой колонки:

```
vleComponent.Cells[0, ARow]
```

Результат будет записан по указателю на структуру типа PPropInfo, в которой находятся параметры указанного свойства. Если этот указатель равен nil, значит, свойство не найдено, то есть, по-видимому, произошла какая-то ошибка с определением выделенного компонента или с именем свойства. В этом случае просто выходим из процедуры.

Если параметры свойства определены верно, то можно приступать к обновлению свойства. Зачем мы определяли параметры? В объектном инспекторе все

строки воспринимаются как текст. Но у компонента могут быть и целочисленные свойства (например, left, top, width, height и т. д.) или свойства с другими типами данных. Если попытаться в эти свойства занести строку, произойдет ошибка. Именно поэтому перед обновлениями мы получаем параметры свойства, чтобы в зависимости от его типа правильно обновлять значения.

Тип свойства находится в поле PropInfo^.PropType^.Kind. Его возможные значения уже знакомы нам, ведь при загрузке мы использовали это поле, но в другой структуре. Так как в объектный инспектор мы загрузили только числа (tkInteger) и строки (tkLString), то проверяем свойство Kind только на соответствие этим типам, других в объектном инспекторе просто нет.

Для обновления свойства используется функция SetOrdProp. Ей нужно передать три параметра.

- Указатель на компонент, свойство которого надо изменить. В нашем случае это выделенный компонент таблицы.
- Указатель типа PPropInfo, который содержит свойства переменной, в том числе и ее имя.
- Значение, которое должно зависеть от типа. Если изменяется число, то этот параметр должен быть числом, если изменяется строка, то и параметр должен быть строковым, и т. д.

После изменения свойства заставляем выделенный компонент заново прорисоваться на случай, если изменилось значение свойства, которое должно отображаться на самом компоненте, например имя таблицы.

ПРИМЕЧАНИЕ

Исходный код рассмотренного в этой главе примера находится на компакт-диске в каталоге Sources\ch04\2-Database Modeler.

Хотя рассмотренный пример довольно сложен, в реальности он очень гибок и эффективен; удивительно, почему в файле помощи все эти функции не документированы? Первое время я даже боялся их использовать. Раз функции не документированы, значит, они могут измениться в будущих версиях Delphi или вообще исчезнуть. Возможно, изначально они предназначались только для внутреннего использования самой средой разработки.

В Delphi 2005 все функции остались на месте и никуда не исчезли. Вполне возможно, они и в будущем не изменятся.

В качестве домашнего задания хочу предложить вам решить классическую задачу. Почему-то в Интернете очень часто можно встретить вопрос, как обновить то или иное свойство у всех компонентов, если оно у них существует? Через RTTI-функции проблема решается очень просто. Достаточно запросить информацию о свойстве у компонента (с помощью функции GetPropInfo), и если результат будет не нулевой, значит, свойство существует и его можно обновлять. Попробуйте реализовать это в коде самостоятельно.

Графика

За что я люблю графику, так это за то, что она дает простор мыслям и воображению. С точки зрения математики и алгоритмов — это самая лучшая тема для тренировки. А о том, как создать самый быстродействующий код для эффекта размытия, можно спорить часами.

Давайте посмотрим на некоторые алгоритмы, которые позволят вам создать максимально быстродействующее графическое приложение. Конечно же, мы будем использовать GDI (Graphic Device Interface — интерфейс графического устройства), поэтому скорость в любом случае окажется ниже, чем при работе через DirectX. И все же, давайте постараемся «выжать» из программы максимальную производительность, сделав при этом код максимально удобным.

Оптимизация графики на примере построения градиента

Ко мне очень часто приходят вопросы о том, как создать быстрый алгоритм градиентной заливки. В последнее время подобных вопросов стало еще больше. Мне кажется, это связано с тем, что в последних версиях Windows система сама очень часто использует этот эффект.

Для реализации примера я создал небольшое приложение, в котором можно выбирать любые цвета и направления заливки. Универсальность в графике, как правило, означает снижение производительности, но мы создадим такой алгоритм, на быстродействии которого возможность выбора направления заливки не слишком скажется.

Итак, создадим новое приложение и на форме расположим два компонента TShape. По щелчку на этих компонентах будет отображаться окно выбора цвета. Помимо этого нам понадобится два компонента типа TRadioButton, чтобы можно было выбрать направление заливки (рис. 4.3).

Рис. 4.3. Форма будущей программы

Градиент будет создаваться только один раз во время изменения параметров (цвета или направления заливки). В этот момент мы должны пересчитать цветовой переход и отобразить его на экране. Кроме того, функция пересчета должна вызываться всякий раз, когда окно перемещается по экрану или перекрывается, то есть при любой перерисовке формы.

Даже при перемещении окна Windows заставляет его перерисовываться, а не сохраняет/восстанавливает. Мы должны сами запоминать все параметры окна.

Чтобы не приходилось пересчитывать лишний раз заливку, лучше всего сохранять ее в компоненте типа TImage или в объекте TBitmap. Я выбрал первый вариант, потому что он требует меньшего объема кода. Таким образом, мы пересчитываем заливку, сохраняем ее в битовом массиве компонента, а в случае, если потребуется перерисовать форму, компонент будет перерисован автоматически без пересчета цветового перехода.

Теперь посмотрим на функцию расчета градиента (листинг 4.6). Если вы читали книгу «Delphi в шутку и всерьез: что умеют хакеры», то приемы оптимизации, которые мы здесь используем, будут вам понятны без дополнительных пояснений. Для тех, кто не читал эту книгу, придется сделать некоторые комментарии.

Листинг 4.6. Функция расчета цветового перехода и сохранения результата в TImage

```
procedure TBackgroundForm.RepaintPicture;
var
 difr,difb,difg,sr,sg,sb:integer;
 x1,x2,y1,y2,h,w,l:integer;
 c:Real;
 u:longint;
 sc,ec,n:longint;
begin
  // Сохранение основных значений в локальных переменных
  sc := Shape1.brush.Color;
  ec := Shape2.brush.Color;
  h := BGImage.Height;
  w := BGImage.Width;
  n := w;

  if VertButton.Checked then
   c := h / n
  else
   c := w / n;

  // Предварительные расчеты

  // Определение каждой составляющей
  sr := sc and $FF;
  sg := (sc shr 8) and $FF;
  sb := (sc shr 16) and $FF;

  // Определение изменения
  difr := (ec and $FF) - sr;
  difg := ((ec shr 8) and $FF) - sg;
  difb := ((ec shr 16) and $FF) - sb;
  BGImage.canvas.pen.style := psclear;
```

```
BGImage.canvas.brush.style := bssolid;
   // Цикл создания градиента
   for l := 0 to n-1 do
    begin
     if VertButton.Checked then
      begin
       xl := 0;
       x2 := w;
       yl := round(l*c);
       y2 := round(yl + c)+2;
      end
     else
      begin
       yl := 0;
       y2 := h;
       xl := round(l*c);
       x2 := round(xl + c)+2;
      end;
     u := rgb((l * difr) div (n-1)+sr,
          (l * difg) div (n-1)+sg,
          (l * difb) div (n-1)+sb);
      BGImage.canvas.Brush.color := u;
      BGImage.canvas.rectangle(xl,yl,x2,y2);
    end;
end;
```

Весь код можно разбить на три части.

1. *Сохранение основных параметров в локальных переменных процедуры.* Для получения доступа к глобальным переменным и конструкциям типа *Объект.Свойство1.Свойство2* требуется гораздо больше времени, чем для получения доступа к локальным переменным, что объясняется рядом причин.

   - При использовании глобальных переменных переходы между ячейками в памяти слишком большие. Локальные же переменные хранятся в стеке и доступ к ним происходит намного быстрее. Да и компилятор сможет лучше оптимизировать код, например на время всех расчетов поместив наиболее часто используемое значение в регистр.

   - Если использовать конструкцию типа *Объект.Свойство1.Свойство2*, то компилятору придется создавать в исполняемом файле несколько строк кода. Сначала нужно определить, где в памяти находится объект, затем найти его первое свойство и только после этого станет возможно найти второе. Так как цвета, которые задаются в цепочке `Shapel.brush.Color`, должны использоваться часто, то вполне логично один раз обратиться к этой цепочке, дабы сохранить значение в локальной переменной, а потом использовать его сколь угодно часто.

2. *Предварительные расчеты.* Для построения градиента понадобится цикл, который будет выполняться достаточно много раз, а если точнее, то количество итераций должно соответствовать ширине области градиента (компонента `TImage`). Каждый раз выполнять сложные операции достаточно накладно для процессора, поэтому самым слабым звеном алгоритма является именно этот

цикл. Чтобы цикл выполнялся максимально быстро, необходимо заранее сделать предварительные расчеты, в которые должно войти получение трех цветовых составляющих и определение изменения каждого цвета.

- Начальное значение цвета находится в свойстве `Shape1.brush.Color` (он был сохранен в переменной `sc`). Нам необходимо знать каждую составляющую этого цвета (красный, зеленый и синий), поэтому разбиваем цвет на составляющие. Если вы не догадались, как это сделать, поговорим об это чуть позже.
- Определение изменения каждого цвета (красный, зеленый и синий) происходит на каждом шаге и полученные значения сохраняются в переменных `difr` (изменение красного), `difg` (изменение зеленого) и `difb` (изменение синего). Определив смещение, достаточно будет только умножить его на номер текущего шага и прибавить начальное значение, чтобы получить результат.

3. *Запуск цикла вычисления размеров прямоугольника прорисовки*. После определения размеров определяем цвет прямоугольника, собирая получившиеся цветовые составляющие с помощью функции `rgb`. Остается только нарисовать очередной прямоугольник и перейти к следующему шагу.

Теперь посмотрим, как происходит разбиение цвета на составляющие. Для этого используются логические операции, которые выполняются максимально быстро.

Цвет в Delphi храниться в формате AABBGGRR. Каждый символ — это шестнадцатеричное число, которое может изменяться от 0 до 15 (в шестнадцатеричном исчислении от 0 до F). Первые два числа зарезервированы и всегда равны нулю. Далее идут пары чисел, определяющие синий (BB), зеленый (GG) и, наконец, красный (RR) цвета. Латинские буквы R, G и B вместе составляют аббревиатуру названия цветовой модели RGB (Red, Green, Blue — красный, зеленый, голубой).

Чтобы определить красную составляющую, необходимо побитно сравнить с помощью операции `and` цвет со значением `$000000FF` (или просто `$FF`). Число 000000FF является маской. Там, где указаны нули, при сравнении с другим числом тоже будут нули. Там, где стоят символы FF, значение не изменится.

Например, допустим, выполняется операция `and` со значением цвета `$0045C7A6` и маской `$000000FF`. В результате получаем результат `$000000A6`. Как видите, кроме последних двух символов, которые отвечают за красный цвет, получились нули. Таким образом мы получили красную составляющую.

Чтобы получить зеленый цвет, необходимо сдвинуть число `$0045C7A6` вправо на 8 бит, то есть как раз на две последние цифры. Это делается с помощью операции `shr`. Например, после выполнения операции `$0045C7A6 shr 16` получим число `$0045C7`. Последние две цифры, отвечающие за красный цвет, просто исчезают. Теперь побитно сравнив результат с числом `$000000FF`, получим `$0000C7`, то есть зеленую составляющую цвета. Таким же образом определяется и синяя составляющая.

Теперь посмотрим, как определяется изменение цвета (переменные difr, difg и difb), например, для красного цвета:

```
difr := (ec and $FF) - sr;
```

Переменная ec содержит значение конечного цвета в цветовом переходе. Конструкция (ec and $FF)определяет красную составляющую этого цвета. Из этого значения вычитаем начальное значение красного цвета (переменная sr) и получаем искомый результат.

Как еще оптимизировать код? Слабым звеном всегда является цикл, потому что итераций в цикле много, и если умножить время каждой итерации на их количество (у нас количество итераций определяется шириной окна), получим существенную нагрузку на процессор.

Листинг 4.7. Оптимизированный цикл

```
if VertButton.Checked then
 begin
  x1 := 0;
  x2 := w;
  // Цикл для вертикального градиента
  for l := 0 to n-1 do
   begin
    y1 := round(l*c);
    y2 := round(y1 + c)+2;

    u := rgb((l * difr) div (n-1)+sr,
        (l * difg) div (n-1)+sg,
        (l * difb) div (n-1)+sb);
    BGImage.canvas.Brush.color := u;
    BGImage.canvas.rectangle(x1,y1,x2,y2);
   end
  end;
 else
  begin
   y1 := 0;
   y2 := h;
   // Цикл для горизонтального градиента
   for l := 0 to n-1 do
    begin
     x1 := round(l*c);
     x2 := round(x1 + c)+2;

     u := rgb((l * difr) div (n-1)+sr,
        (l * difg) div (n-1)+sg,
        (l * difb) div (n-1)+sb);
     BGImage.canvas.Brush.color := u;
     BGImage.canvas.rectangle(x1,y1,x2,y2);
    end;
   end;
```

Простейший вариант повышения производительности — вынести оператор сравнения за пределы цикла. В этом случае на каждой итерации будет экономиться одна операция, что уже существенно. В результате, который представлен в листинге 4.7, у нас получилось два цикла вместо одного, но за счет того, что

циклы стали проще и содержат меньше операций, они выполняются быстрее. Помимо логической операции за пределы цикла я вынес команды, которые не изменяются внутри цикла. Для вертикальной заливки — это задание ширины, а для горизонтальной заливки — высоты прямоугольника. А это уже экономия трех операций, и если умножить их на количество итераций, получим экономию в сотни процессорных тактов.

Таким образом, мы получили достаточно высокую производительность. Можно упростить математические операции, но это не даст большого эффекта. Гораздо больший эффект дал бы перевод кода на встроенный ассемблер. Но это уже из области супероптимизации, напоминающей шлифование уже гладкой поверхности.

Пока что мы оптимизировали алгоритм и код, не пытаясь жертвовать качеством цветового перехода. У нас цикл выполняется многократно только для того, чтобы переход получился максимально плавным. В идеальном случае за одну итерацию мы рисуем прямоугольник высотой (или шириной при вертикальной заливке) 1 пиксел, то есть одну строку (или колонку при горизонтальной заливке). Но если рисовать за одну итерацию две строки, человеческий глаз не заметит особой разницы, потому что переход цвета останется плавным.

Рис. 4.4. Пример со ступенчатым градиентом

За количество проходов у нас отвечает переменная n, и по умолчанию в нее записывается значение ширины окна. Если это число разделить на два, то, как мы только что выяснили, качество не слишком изменится:

```
n := w div 2;
```

Снижение качества позволяет добиться высокой скорости. Если разделить переменную n на 10, для программы понадобиться в 10 раз меньше циклов. Если

каждый цикл слишком накладен для процессора, то повышение производительности программы окажется существенным.

Для моего глаза незаметны изменения при делении даже на 10. Но у меня не стопроцентное зрение, поэтому более зоркий глаз, вероятно, увидит «лесенку» в цветовом переходе. Чтобы ее увидели все, нужно разделить n на большее чем 10 число, например на 30 (рис. 4.4). Ваша задача подобрать такое число, которое позволит добиться необходимой производительности при достаточном качестве.

ПРИМЕЧАНИЕ

Исходный код рассмотренного в этой главе примера находится на компакт-диске в каталоге Sources\ch04\Gradient.

Попиксельный анализ изображения

В примере с градиентом нам приходилось рисовать изображение построчно, и в самом худшем случае для создания целого изображения требовалось выполнять цикл столько раз, сколько строк у изображения. А если пришлось бы рисовать каждый пиксел изображения? В этом случае число итераций цикла равнялось бы количеству строк, умноженному на количество колонок. Таким образом, для построения изображения размером 800 × 600 цикл надо было бы выполнить 480 000 раз. Почти полмиллиона – достаточно большая цифра, ощутимая даже для самого быстрого процессора.

Предположим, нужно создать эффект размытия, для реализации которого берется одна точка и ее цвет вычисляется по определенной формуле (зависит от варианта размытия) на основе цвета соседних точек. В простейшем случае значения цветов округляются. Качество алгоритма зависит от количества анализируемых соседних точек.

Если для реализации алгоритма использовать свойство Pixels объекта TCanvas, алгоритм будет выглядеть следующим образом:

```
var
 i, j: Integer;
begin
 for i:=0 to Image.Width do
  for j:=0 to Image.Height do
   begin
    Расчет цвета пиксела Image.Canvas.Pixels[i,j]
    на основе пикселов:
    Image.Canvas.Pixels[i-1,j],
    Image.Canvas.Pixels[i-1,j-1],
    Image.Canvas.Pixels[i,j+1],
    Image.Canvas.Pixels[i+1,j],
   end;
end;
```

Даже с учетом того, что в этом примере учитываются только четыре соседних пиксела, программа будет выполняться чрезвычайно медленно, и основной «вклад» в это вносит свойство Pixels.

Положение спасает метод ScanLine объекта TBitmap. Давайте добавим в код, представленный в предыдущем разделе, средства рисования линий и размытия всего рисунка. Для этого нужно преобразовать главную форму так, чтобы пользователь мог выбирать тип и цвет узора, рисуемого поверх градиента (рис. 4.5).

Рис. 4.5. Окно программы рисования на градиенте

В этом примере после создания градиента мы должны поверх него нарисовать узор и размыть его одним из методов размытия. Нарисовать узор несложно, так что эту тему мы опустим, а код, реализующий алгоритм размытия, можно увидеть в листинге 4.8.

Листинг 4.8. Процедура размытия изображения

```
procedure TBackgroundForm.SplitBlur(var clip: TBitmap; Amount: integer);
var
 p0,p1,p2:PByteArray;
 cx,x,y: Integer;
 Buf: array[0..3,0..2]of Byte;
begin
 if Amount=0 then
  exit;

 for y:=0 to clip.Height-1 do
  begin
   p0:=clip.scanline[y];
   if y-Amount<0 then
    p1:=clip.scanline[y]
   else
    p1:=clip.ScanLine[y-Amount];

   if y+Amount<clip.Height then
    p2:=clip.ScanLine[y+Amount]
   else
    p2:=clip.ScanLine[clip.Height-y];

   for x:=0 to clip.Width-1 do
    begin
     if x-Amount<0
```

```
      then cx:=x
     else
      cx:=x-Amount;
     Buf[0,0]:=p1[cx*3];
     Buf[0,1]:=p1[cx*3+1];
     Buf[0,2]:=p1[cx*3+2];
     Buf[1,0]:=p2[cx*3];
     Buf[1,1]:=p2[cx*3+1];
     Buf[1,2]:=p2[cx*3+2];

     if x+Amount<clip.Width then
      cx:=x+Amount
     else
      cx:=clip.Width-x;

     Buf[2,0]:=p1[cx*3];
     Buf[2,1]:=p1[cx*3+1];
     Buf[2,2]:=p1[cx*3+2];
     Buf[3,0]:=p2[cx*3];
     Buf[3,1]:=p2[cx*3+1];
     Buf[3,2]:=p2[cx*3+2];
     p0[x*3]:=(Buf[0,0]+Buf[1,0]+Buf[2,0]+Buf[3,0])shr 2;
     p0[x*3+1]:=(Buf[0,1]+Buf[1,1]+Buf[2,1]+Buf[3,1])shr 2;
     p0[x*3+2]:=(Buf[0,2]+Buf[1,2]+Buf[2,2]+Buf[3,2])shr 2;
    end;
  end;
end;
```

Код пока не слишком оптимизирован, зато прост для понимания. Итак, сначала посмотрим на параметры, которые передаются процедуре:

- clip — переменная типа TBitmap, то есть размываемое растровое изображение;
- Amount — коэффициент размытия; чем больше это число, тем более размытым будет результат.

Сначала проверяем, равно ли нулю число Amount, и если равно, выходим из процедуры. При таком значении исходное изображение и результат будут идентичными, поэтому нет смысла тратить время на обработку изображения.

После этого запускается цикл, который будет выполняться от 0 до количества строк в растровом изображении. На каждой итерации будем обрабатывать целую строку. Таким образом, мы значительно экономим на итерациях, поскольку обрабатываем изображение не попиксельно, а построчно.

В процедуре есть три переменных p0, p1, и p2. Все они имеют тип PByteArray, то есть массив из чисел типа Byte. В p0 записываем массив данных о текущей строке:

```
p0:=clip.scanline[y];
```

В массив p1 записываем строку графических данных y (текущая строка) минус число Amount. В массив p2 записываем данные строки y плюс Amount. Так как во время записи в массивы p1 и p2 приходится обращаться к строкам, которые отличаются от текущей, необходима уверенность, что мы не выйдем за границы растрового изображения.

Таким образом, обращаясь к данным изображения только три раза за цикл, получаем целые строки. Это намного быстрее, чем анализировать строку через свойство Pixels. Получив три массива данных, запускаем второй цикл, который выполняется от 0 до количества пикселов в ширину изображения:

```
for x:=0 to clip.Width-1 do
```

Здесь уже идет обработка полученных буферов. Их данные можно как читать, так и изменять.

Дальнейший разбор листинга не имеет особого смысла, потому что он достаточно прост. Надо заметить, что алгоритм по-прежнему не оптимален. На каждом шаге нам приходится трижды обращаться к растровому изображению, чтобы получить данные строк. Есть возможность сократить обращение до одного. Правда, в этом случае придется создавать чуть больше буферов для хранения строк, точнее, их может быть и три, но в этом случае значение Amount должно быть равно единице.

Надеюсь, что этих подсказок достаточно. Попробуйте сами оптимизировать алгоритм и сделать его разумно быстрым. Почему «разумно»? Потому что не стоит погружаться в дебри ассемблера, используйте только Delphi.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch04\ScanLine. Оптимизированный пример вы должны создать сами.

Цветовая панель

Если пользователь вашей программы должен иметь возможность выбирать цвет, то для реализации этой возможности в Delphi есть только компонент ColorDialog, представляющий собой стандартное диалоговое окно выбора цвета. То, что окно стандартное, — это хорошо, но при частой смене цветов использовать его неудобно. Недаром во многие графические программы встраивают палитры, на которых представлены основные цвета. Например, в программе Adobe Photoshop есть палитра, позволяющая выбирать цвета с помощью ползунков и специальной шкалы, образованной цветовыми переходами (рис. 4.6). Такая палитра занимает мало места на экране и удобна в использовании.

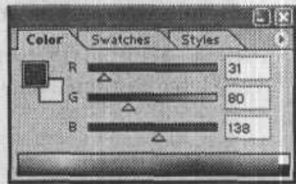


Рис. 4.6. Палитра выбора цвета в программе Adobe Photoshop

К сожалению, такая палитра до сих пор не входит в стандартную поставку среды разработки Delphi 2005, и приходится использовать компоненты сторонних

разработчиков. Давайте исправим эту ситуацию, и напишем быструю, удобную и универсальную панель выбора цвета (рис. 4.7).

Рис. 4.7. Такую панель нам предстоит создать

Для реализации панели я решил создать новый визуальный компонент, который вы сможете установить на любую форму своего приложения. Таким образом, мы попрактикуемся и в программировании графики, и в создании компонентов.

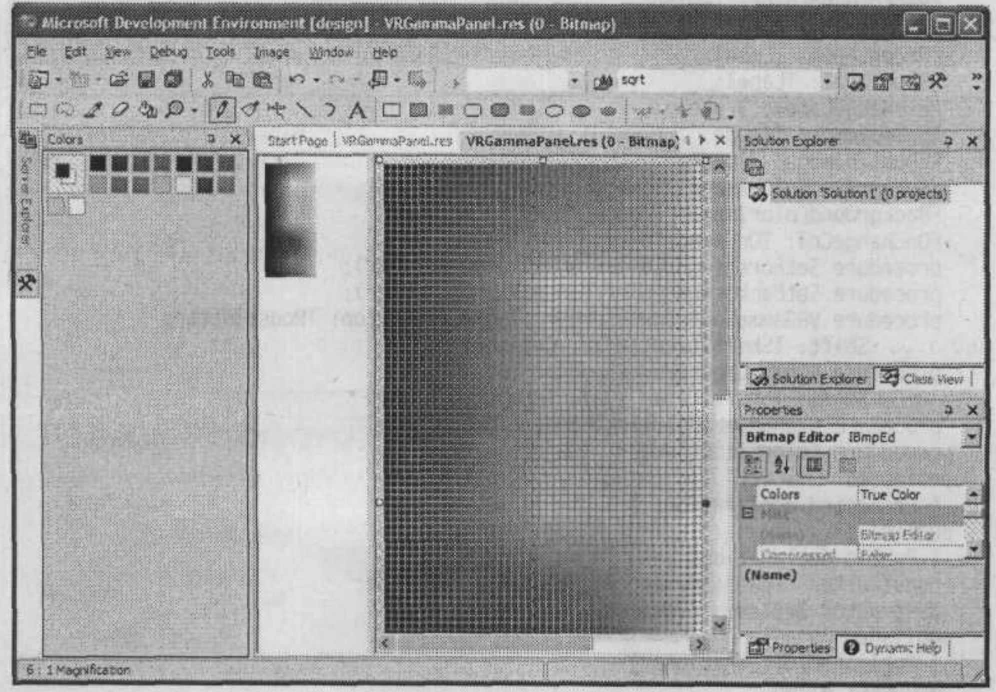


Рис. 4.8. Редактирование ресурсов в Visual Studio .NET

Как воспроизвести в программе цветовую гамму? Можно программно по определенному алгоритму рисовать на компоненте `TImage`, но зачем? Поскольку гамма не будет изменяться во время выполнения программы, вполне логично заранее создать растровый рисунок и загружать его в компонент `TImage`. Так и сделаем.

Создайте res-файл в редакторе ресурсов. Тут есть одна проблема — в поставке Delphi имеется программа Image Editor, которая позволяет создавать ресурсы с изображениями, значками и указателями мыши. Все бы хорошо, но программа поддерживает только 256 цветов, а для нас этого мало, поэтому придется искать что-то другое. Для создания полноцветного (True Color) изображения в ресурсе мне пришлось использовать программу Microsoft Visual Studio .NET 2003 (рис. 4.8).

Итак, создадим новый компонент, который назовем `VRGammaPanel`, а в качестве его предка выберем компонент `TWinControl`. Объявление нашего компонента показано в листинге 4.9. Внимательно изучите это объявление.

Листинг 4.9. Объявление панели выбора цвета

```
TOnChangeColor = procedure(Sender: TObject;
        Foreground, Background: TColor) of object;

TVRGammaPanel = class(TWinControl)
private
 FForegroundColor: TColor;
 FBackgroundColor: TColor;
 FRedLabel: TLabel;
 FGreenLabel: TLabel;
 FBlueLabel: TLabel;
 FExchangeLabel: TLabel;
 FGammaImage: TImage;
 FChoosedImage: TImage;
 FForegroundColorImage: TImage;
 FBackgroundColorImage: TImage;
 FOnChangeCol: TOnChangeColor;
 procedure SetForegroundColor(const Value: TColor);
 procedure SetBackgroundColor(const Value: TColor);
 procedure VRGammaMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
 procedure VRGammaMouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
 procedure ExchangeLabelClick(Sender: TObject);
 procedure ColorClick(Sender: TObject);
protected
 { Protected declarations }
public
 { Public declarations }
 constructor Create(AOwner: TComponent); override;
 destructor Destroy; override;
published
 { Published declarations }
 property Align;
 property OnChangeColor: TOnChangeColor read FOnChangeCol
```

```
    write FOnChangeCol;
  property ForegroundColor: TColor read FForegroundColor
    write SetForegroundColor default clBlack;
  property BackgroundColor: TColor read FBackgroundColor
    write SetBackgroundColor default clWhite;
end;
```

Посмотрим, что у нас есть в объявлении.

- Компоненты FBackgroundColor и FForegroundColor будут использоваться для хранения выбранных цветов фона и переднего плана.
- Компоненты FRedLabel, FGreenLabel и FBlueLabel будут отображать составляющие текущего выделенного цвета.
- Компонент FExchangeLabel может отображать текст, но главное, что по щелчку на этом компоненте цвета́ фона и переднего плана будут меняться местами.
- В компоненте FGammaImage, имеющем тип TImage, будет представлена вся цветовая гамма.
- Так как в гамме каждый цвет представлен только одним пикселом, реально увидеть его цвет очень сложно, поэтому мы используем компонент FChoosedImage, призванный воспроизводить цвет точки, на которой находится указатель мыши.
- Компоненты FForegroundColorImage и FBackgroundColorImage будут отображать текущие цвета фона и переднего плана.

После объявления всех типов в разделе implementation подключим файл ресурсов, в котором находится изображение гаммы. У меня это файл VRGammaPanel.res:

```
{$R VRGammaPanel.res}
```

В конструкторе Create нам необходимо создать все изображения и подписи, а затем расположить их поверх компонента (листинг 4.10).

Листинг 4.10. Конструктор компонента

```
constructor TVRGammaPanel.Create(AOwner: TComponent);
begin
  inherited;
  Width := 65;
  Height := 250;

  FForegroundColor := clBlack;
  FBackgroundColor := clWhite;

  // Создаем компонент отображения красной составляющей выбранного цвета
  FRedLabel := TLabel.Create(self);
  FRedLabel.Parent := self;
  FRedLabel.Top := 152;
  FRedLabel.Left := 5;
  FRedLabel.Caption := 'R:';

  // Создаем компонент отображения зеленой составляющей выбранного цвета
  FGreenLabel := TLabel.Create(self);
```

Листинг 4.10 (продолжение)

```
FGreenLabel.Parent := self;
FGreenLabel.Top := 168;
FGreenLabel.Left := 5;
FGreenLabel.Caption := 'G:';

// Создаем компонент отображения синей составляющей выбранного цвета
FBlueLabel := TLabel.Create(self);
FBlueLabel.Parent := self;
FBlueLabel.Top := 184;
FBlueLabel.Left := 5;
FBlueLabel.Caption := 'B:';

// Создаем изображение цветовой гаммы
FGammaImage := TImage.Create(self);
FGammaImage.Parent := self;
FGammaImage.Stretch := False;
FGammaImage.AutoSize := True;
FGammaImage.Top := 4;
FGammaImage.Left := 5;
FGammaImage.Width := 55;
FGammaImage.Height := 105;
FGammaImage.OnMouseDown := VRGammaMouseDown;
FGammaImage.OnMouseMove := VRGammaMouseMove;
FGammaImage.Picture.Bitmap.LoadFromResourceName(HInstance, 'COLORS');
FGammaImage.Cursor := crCross;

// Создаем компонент отображения выбранного цвета
FChoosedImage := TImage.Create(self);
FChoosedImage.Parent := self;
FChoosedImage.Top := 200;
FChoosedImage.Left := 12;
FChoosedImage.Width := 30;
FChoosedImage.Height := 30;
FChoosedImage.Canvas.Brush.Color := FForegroundColor;
FChoosedImage.Canvas.Brush.Style := bsSolid;
FChoosedImage.Canvas.FillRect(Rect(0, 0, FChoosedImage.Width,
      FChoosedImage.Height));

// Создаем компонент отображения выбранного цвета переднего плана
FForegroundColorImage := TImage.Create(self);
FForegroundColorImage.Left := 5;
FForegroundColorImage.Top := 105;
FForegroundColorImage.Width := 25;
FForegroundColorImage.Height := 25;
FForegroundColorImage.Parent := self;
FForegroundColorImage.Canvas.Brush.Color := clBlack;
FForegroundColorImage.Canvas.Brush.Style := bsSolid;
FForegroundColorImage.Canvas.FillRect(Rect(0, 0, FChoosedImage.Width,
      FChoosedImage.Height));
FForegroundColorImage.Hint := 'Цвет фона';
FForegroundColorImage.ShowHint := True;
```

```
  FForegroundColorImage.Name := 'imForeground';
  FForegroundColorImage.OnClick := ColorClick;

  // Создаем компонент отображения выбранного цвета фона
  FBackgroundColorImage := TImage.Create(self);
  FBackgroundColorImage.Left := 25;
  FBackgroundColorImage.Top := 120;
  FBackgroundColorImage.Height := 25;
  FBackgroundColorImage.Width := 25;
  FBackgroundColorImage.Parent := self;
  FBackgroundColorImage.Canvas.Brush.Color := FBackgroundColor;
  FBackgroundColorImage.Canvas.Brush.Style := bsSolid;
  FBackgroundColorImage.Canvas.FillRect(Rect(0, 0, FChoosedImage.Width,
      FChoosedImage.Height));
  FBackgroundColorImage.Hint := 'Цвет карандаша';
  FBackgroundColorImage.ShowHint := True;
  FBackgroundColorImage.Name := 'imBackround';
  FBackgroundColorImage.OnClick := ColorClick;

  // Компонент TLabel, по щелчку на котором цвета меняются местами
  FExchangeLabel := TLabel.Create(self);
  FExchangeLabel.Left := 7;
  FExchangeLabel.Top := 132;
  FExchangeLabel.Caption := '<>';
  FExchangeLabel.Hint := 'Поменять цвета';
  FExchangeLabel.OnClick := ExchangeLabelClick;
  FExchangeLabel.ShowHint := True;
  FExchangeLabel.Parent := self;
end;
```

Вначале создаются компоненты типа TLabel, в которых будут отображаться числовые значения составляющих каждого цвета. Тут ничего сложного нет, просто обеспечиваются создание, позиционирование поверх компонента и задание текстового заголовка.

Создав компонент TImage для отображения цветовой гаммы и расположив его сверху компонента, устанавливаем обработчики событий OnMouseDown (нажатие кнопки мыши на компоненте) и OnMouseMove (движение мыши на компоненте):

```
FGammaImage.OnMouseDown := VRGammaMouseDown;
FGammaImage.OnMouseMove := VRGammaMouseMove;
```

Когда пользователь нажимает кнопку мыши на изображении гаммы, наша задача выяснить, какой цвет находится в точке нажатия, и в зависимости от того, какая кнопка нажата, изменить текущий цвет переднего плана (нажата левая кнопка) или цвет фона (нажата правая кнопка).

Следующим этапом загружаем изображение гаммы в компонент TImage:

```
FGammaImage.Picture.Bitmap.LoadFromResourceName(
  HInstance, 'COLORS');
```

В моем случае изображение гаммы сохранено в ресурсе под именем 'COLORS'.

Когда указатель мыши находится на гамме, компонент FChoosedImage должен окрашиваться цветом текущего пиксела. Таким образом будет создаваться впечатление просмотра цветового образца в увеличенном виде.

Далее выполняется создание компонента FChoosedImage типа TImage. Задаем его позицию так, чтобы он располагался внизу панели. В качестве цвета по умолчанию берем цвет переднего плана и окрашиваем компонент:

```
FChoosedImage.Canvas.Brush.Color := FForegroundColor;
FChoosedImage.Canvas.Brush.Style := bsSolid;
FChoosedImage.Canvas.FillRect(Rect(0, 0,
        FChoosedImage.Width, FChoosedImage.Height));
```

Следующими создаются компоненты FForegroundColorImage и FBackgroundColorImage, в которых отображаются выбранные цвета переднего плана и фона. Позиционируем их так, чтобы они перекрывали друг друга, как это делается в большинстве графических редакторов (например, в Adobe Photoshop). Окрашиваем их в цвета, которые мы выбрали по умолчанию (черный и белый соответственно) и задаем подсказки (свойство Hint), чтобы при наведении указателя мыши пользователь видел, к чему относится цвет.

Для обоих компонентов перехватываем событие OnClick. По щелчку мышью на этих компонентах на экран будет выводиться стандартное окно выбора цвета. Да, мы пишем панель, но для выбора системных цветов основной палитры все же удобнее использовать стандартное окно. В качестве обработчика события установим процедуру ColorClick. Как обработчик будет определять, на каком из компонентов пользователь щелкнул мышью? Забегая вперед, скажу, что у компонентов специально для этого задается свойство Name.

Напоследок создаем компонент FExchangeLabel типа TLabel. У него нам необходим обработчик события OnClick, в котором будем менять цвета переднего плана и фона местами.

В деструкторе Destroy, следуя хорошему тону программирования, уничтожим все компоненты, которые создали программно в конструкторе. Этот код я приводить не буду, дабы сэкономить место для более интересных вещей. Единственное, на что я хочу обратить ваше внимание, — уничтожать объекты необходимо до выполнения команды inherited. Эта команда вызывает деструктор предка, в котором сам компонент уничтожается, и последующие действия по уничтожению дочерних компонентов могут привести к сбою в программе.

Давайте посмотрим, как можно реализовать обработчик ColorClick, который обрабатывает щелчок на обоих компонентах TImage, воспроизводящих выбранные цвета. Код обработчика показан в листинге 4.11.

Листинг 4.11. Смена текущего цвета с помощью окна диалога

```
procedure TVRGammaPanel.ColorClick(Sender: TObject);
begin
 with TColorDialog.Create(Self) do
  begin
   if Execute then
    begin
     if TImage(Sender).Name='imForeground' then
      SetForegroundColor(Color)
     else
      SetBackgroundColor(Color);
    end;
```

```
    Free;
  end;
end;
```

У создаваемого нами компонента нет отдельной переменной типа TColorDialog, и мы ничего пока не создавали. Да это и не нужно, потому что создавать и уничтожать окно мы будем динамически. Это происходит в первой строке процедуры:

```
with TColorDialog.Create(Self) do
```

Напоминаю, что оператор with..do указывает на то, что последующий код между ключевыми словами begin и end относится к созданному окну диалога. А здесь мы отображаем окно методом Execute, проверяем, выбрал ли пользователь цвет, и если выбрал, то какой компонент сгенерировал сообщение: если это компонент imForeground — изменяем цвет переднего плана, иначе изменяется цвет фона. Для этого используются функции SetForegroundColor и SetBackgroundColor, которые можно увидеть в листинге 4.12.

Листинг 4.12. Смена цветов переднего плана и фона

```
procedure TVRGammaPanel.SetForegroundColor(const Value: TColor);
begin
  FForegroundColor := Value;

  // Перерисовать компонент отображения цвета переднего плана
  FForegroundColorImage.Canvas.Brush.Color := FForegroundColor;
  FForegroundColorImage.Canvas.Brush.Style := bsSolid;
  FForegroundColorImage.Canvas.FillRect(Rect(0, 0,
     FForegroundColorImage.Width, FForegroundColorImage.Height));

  // Если назначен обработчик события, то вызвать его
  if Assigned(FOnChangeCol) then
   FOnChangeCol(Self, FForegroundColor, FBackgroundColor);
end;

procedure TVRGammaPanel.SetBackgroundColor(const Value: TColor);
begin
  FBackgroundColor := Value;

  // Перерисовать компонент отображения цвета фона
  FBackgroundColorImage.Canvas.Brush.Color := FBackgroundColor;
  FBackgroundColorImage.Canvas.Brush.Style := bsSolid;
  FBackgroundColorImage.Canvas.FillRect(Rect(0, 0,
     FBackgroundColorImage.Width, FBackgroundColorImage.Height));

  // Если назначен обработчик события, то вызвать его
  if Assigned(FOnChangeCol) then
   FOnChangeCol(Self, FForegroundColor, FBackgroundColor);
end;
```

Функции идентичны, поэтому рассматривать их будем обе сразу. В первой строке сохраняется цвет в соответствующей переменной. Далее перерисовываем компонент TImage, который отображает цвет, и проверяем, если назначен обработчик события OnChangeCol, то он вызывается.

Последнее, что мы должны сделать, — написать обработчики событий OnMouseMove и OnMouseDown (листинг 4.13).

Листинг 4.13. Обработчики событий мыши

```
procedure TVRGammaPanel.VRGammaMouseMove(Sender: TObject; Shift:
  TShiftState; X, Y: Integer);
var
 col: TColor;
begin
 col := FGammaImage.Picture.Bitmap.Canvas.Pixels[X, Y];

 // Отображаем составляющие текущего цвета в компонентах TLabel
 FRedLabel.Caption := 'R:'+ IntToStr(GetRValue(col));
 FGreenLabel.Caption := 'G:'+ IntToStr(GetGValue(col));
 FBlueLabel.Caption := 'B:'+ IntToStr(GetBValue(col));

 // Окрашиваем изображение FChoosedImage выбранным цветом
 FChoosedImage.Canvas.Brush.Color := col;
 FChoosedImage.Canvas.Brush.Style := bsSolid;
 FChoosedImage.Canvas.FillRect(Rect(0, 0,
   FChoosedImage.Width, FChoosedImage.Height));
end;

procedure TVRGammaPanel.VRGammaMouseDown(
   Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 if Button = mbLeft then
  SetForegroundColor(
      FGammaImage.Picture.Bitmap.Canvas.Pixels[X, Y]);
 if Button = mbRight then
  SetBackgroundColor(
      FGammaImage.Picture.Bitmap.Canvas.Pixels[X, Y]);
end;
```

По событию OnMouseMove вначале определяем цвет пиксела, который находится под указателем мыши. Затем этот цвет разбиваем на составляющие и заполняем значениями соответствующие компоненты TLabel. После этого окрашиваем компонент FChoosedImage типа TImage выбранным цветом.

По щелчку мыши, в зависимости от того, какой именно кнопкой, правой или левой, выполнен щелчок, устанавливаем цвет фона или переднего плана.

Как видите, реализовать такой компонент не составляет особого труда. Немного стараний, и мы получаем универсальное решение, а если потратить еще некоторое время, компонент можно наделить колоссальными возможностями.

Чтобы протестировать пример, установите его в Delphi. Среди свойств вы увидите свойства BackgroundColor и ForegroundColor, с помощью которых можно узнать или изменить текущие цвета фона и переднего плана как во время разработки, так и во время выполнения программы.

ПРИМЕЧАНИЕ

Исходный код компонента и программы рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch04\GammaPanel. А в каталоге Additional\GradientWithGamma можно найти реализацию градиента с помощью панели. Оптимизированный пример вы должны создать сами.

Манипуляции с компонентом

В разделе «RTTI» мы написали пример, позволяющий двигать мышью размещенные на форме компоненты. Но этот пример далек от идеала, поскольку не дает изменять размеры компонентов. В принципе, это сделать не так уж сложно. Нужно просто выполнить не перемещение, а изменение размера компонента, когда пользователь нажимает кнопку мыши на границе компонента.

Однако все просто только на словах. В соответствии со стандартом вокруг выделенного щелчком мыши компонента появляется рамка выделения с маркерами (рис. 4.9), и для изменения размеров компонента перетаскивать нужно именно эти маркеры.

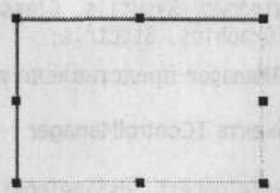


Рис. 4.9. Маркеры изменения размера компонента

В нашем примере, который мы создали в разделе «RTTI», для начала нужно переписать метод перерисовки. Если компонент выделен, то необходимо рисовать по краям маркеры. Кроме того, нужно переделать функцию перетаскивания, чтобы учесть возможность изменения размера. Кода придется писать много, но не торопитесь это делать. А что если в будущем вам понадобится перетаскивать и изменять размеры кнопки, доступа к исходному коду которой у вас не будет или вам будет запрещено менять этот код?

Как решить проблему? Первая мысль, которая приходит в голову, — программно искать компонент и рисовать на нем маркеры. В большинстве случаев это решение сработает, хотя оно и кажется достаточно сложным. А вот в плане универсальности что-либо предложить здесь трудно.

Разработка универсального менеджера компонентов

Недавно мне предложили одну интересную идею, которую я расширил, оптимизировал и улучшил, а теперь предлагаю вам в виде универсального компонента. С его помощью вы сможете перемещать и изменять размеры любого компонента. Давайте рассмотрим процесс создания этого универсального компонента. Я рекомендую вам повторить мои действия, чтобы вы самостоятельно создали свою реализацию. Конечно, можно просто взять мой код и использовать в своих проектах, но возможности своей реализации вы с легкостью сможете наращивать.

Сначала рассмотрим саму идею. Все основывается на компоненте с прозрачной рабочей областью, в качестве основы для которого можно взять компонент

TCustomControl, обладающий всеми необходимыми нам свойствами. Чтобы иметь возможность перемещать или изменять размеры какого-либо компонента на форме, мы как бы накрываем его своим компонентом, а затем перетаскиваем уже наш компонент, который автоматически перемещает и изменяет размеры исходного компонента.

Зачем нужно накрывать? Чтобы не трогать реализации существующих компонентов, а рисовать необходимые маркеры на своем компоненте. Всю функциональность должен обеспечивать наш компонент, код других компонентов изменять нельзя.

Итак, создаем новый модуль и новый объект, который назовем TControlManager. В разделе uses нам понадобятся следующие модули:

```
uses ExtCtrls, Messages, Windows, SysUtils, Classes, Math,
  Controls, Forms, Menus, Graphics, StdCtrls;
```

Объявление объекта TControlManager представлено в листинге 4.14.

Листинг 4.14. Объявление объекта TControlManager

```
type
  TDragStyle = (dsMove, dsSizeTopLeft, dsSizeTopRight, dsSizeBottomLeft,
     dsSizeBottomRight, dsSizeTop, dsSizeLeft, dsSizeBottom, dsSizeRight);

  TMessageType = (fmDown, fmUp);

  TControlManager = class(TCustomControl)
   private
     ptDragOffset: TPoint;
     bDragging: boolean;
     FControlsList: TList;
     FDragStyle: TDragStyle;
     FDragRect: TRect;
   protected
     procedure Paint; override;
     procedure WMEraseBkgnd(var Message: TWMEraseBkgnd);
          message WM_ERASEBKGND;
     procedure CreateParams(var Params: TCreateParams); override;
     procedure MouseDown(Button: TMouseButton; Shift: TShiftState;
          X, Y: Integer); override;
     procedure MouseUp(Button: TMouseButton; Shift: TShiftState;
          X, Y: Integer); override;
     procedure MouseMove(Shift: TShiftState; X, Y: Integer); override;
     procedure ReSendMessage(FMsg: TMessageType; Button:
        TMouseButton; Shift: TShiftState; X, Y: Integer);
     function GetModifiedRect(XPos, YPos: integer): TRect;
     procedure DrawDraggingRect(XPos, YPos: integer; ShowBox: boolean);
     procedure UpdateChildren(aLeft, aTop, aWidth, aHeight: Integer);
   public
     constructor Create(AOwner: TComponent); override;
     destructor Destroy; override;
     procedure AddControl(clControl: TControl);
     procedure ClearControls;
   published
  end;
```

Разберемся, что у нас тут есть и для чего все это нужно. Так будет проще писать реализацию.

В самом начале, до объявления компонента TControlManager, для удобства у нас объявлены два новых типа.

- TDragStyle — этот тип отвечает за выполняемое действие (перемещение, изменение размера в ширину вправо, изменение размера в ширину влево и т. д.). Другими словами, этот тип определяет, за какую точку компонента пользователь «ухватился» мышью и в какую сторону он тянет компонент.
- TMessageType — этот тип предназначен для того, чтобы удобней было передавать события мыши нашему компоненту (поскольку мы накрываем чужой компонент, все события мыши должны идти нашему компоненту). В принципе, можно было бы обойтись и без него, но так удобней и красивее.

Теперь рассмотрим объект TControlManager, который является потомком от TCustomControl. У него нам понадобятся следующие свойства.

- ptDragOffset — в примере из раздела «RTTI» начальную точку перетаскивания мы хранили в переменных iStartDraggingX и iStartDraggingY. Здесь я решил использовать переменную типа TPoint, более удобную для решения этой задачи.
- bDragging — если переменная равна true, это означает, что нажата левая кнопка мыши и при перемещении указателя мыши компонент нужно двигать.
- FControlsList — указатель на список всех накрытых компонентов. Да, именно список. Поскольку пример должен быть универсальным, мы должны обеспечить возможность работы сразу с целым списком компонентов.
- FDragStyle — в эту переменную будем записывать одно из значений типа TDragStyle в качестве идентификатора происходящего действия.
- FDragRect — в этой переменной будет храниться область с новыми размерами и положением компонента. Поскольку пример должен быть универсальным, при перетаскивании или изменении размера мы не будем отображать весь компонент. Если у него сложное содержимое, которое долго прорисовывается, то перетаскивание или изменение его размера может идти рывками. Чтобы этого не происходило, будем двигать только рамку. В ОС Windows так реализовано перетаскивание окон — при перетаскивании можно отображать содержимое окон или только рамку. После завершения перетаскивания окно принимает размеры и положение рамки.

Теперь рассмотрим методы компонента TControlManager. Конечно же, здесь есть конструктор и деструктор для выделения и освобождения памяти. У нас используется объектная переменная FControlsList типа TList, и именно ее нужно создавать в конструкторе и уничтожать в деструкторе:

```
constructor TControlManager.Create(AOwner: TComponent);
begin
 inherited; // Не забываем вызывать конструктор предка
 FControlsList:=TList.Create; // Создание списка
end;

destructor TControlManager.Destroy;
```

```
begin
  FControlsList.Free; // Уничтожение списка
  inherited; // Не забываем вызывать деструктор предка
end;
```

Помимо конструктора и деструктора нам понадобятся открытые методы, с помощью которых можно будет добавлять компоненты в список `FControlsList` (список накрываемых компонентов) и освобождать. Для добавления компонента будем использовать метод `AddControl`, для очистки списка — метод `ClearControls`.

Остальные методы будем рассматривать по мере реализации компонента, а начнем реализацию с метода `AddControl` (листинг 4.15).

Листинг 4.15. Добавление компонента в список

```
procedure TControlManager.AddControl(clControl: TControl);
var
  l, t, w, h: integer;
begin
  // Если компонент уже есть в списке, то выходим
  if (FControlsList.IndexOf(clControl) > -1) then
   exit;

  // Если в списке есть компоненты и у нового компонента другой предок,
  // то очищаем список
  if (FControlsList.Count > 0) and (clControl.Parent <> Parent) then
   ClearControls;

  // Если компонентов в списке нет, то
  if FControlsList.Count = 0 then
   begin
    Parent := clControl.Parent;

    FDragRect := Rect(0, 0, 0, 0);
    Visible := True;
    SetBounds(clControl.Left - 3, clControl.Top - 3,
            clControl.Width + 3, clControl.Height + 3);
   end
  else // Иначе компоненты в списке есть и нужно рассчитать новый размер
   begin
    l := Min(Left, clControl.Left - 2);
    t := Min(Top, clControl.Top - 2);
    w := Max(Left + Width - 3, clControl.Left + clControl.Width) - l + 3;
    h := Max(Top + Height - 3, clControl.Top + clControl.Height) - t + 3;
    SetBounds(l, t, w, h);
   end;

  // Добавляем компонент в список
  FControlsList.Add(clControl);
  FDragStyle := dsMove; /

  // Располагаем компонент поверх остальных
  BringToFront;
  SetFocus;
end;
```

Сначала проверяем, находится ли в списке компонент, который собираются подключить. Если находится, то выходим из метода и не пытаемся продолжать работу. Зачем дважды подключать одно и то же. Чтобы определить, находится ли компонент в списке, используется метод `IndexOf`. Ему нужно передать указатель на объект, и если такой указатель уже есть в списке, то метод вернет его индекс. Если ничего не найдено, то результатом будет −1.

Затем проверяем, какой компонент добавляется в список. Если в списке уже есть компоненты, а в список добавляется компонент с другим предком, то очищаем весь список. Почему? Попробуйте в Delphi выделить два компонента, которые располагаются на разных панелях. Не получилось? И в нашем примере тоже нельзя будет так делать.

Дело в том, что в соответствии с основной идеей, когда компонент добавляется в список, мы должны накрыть его своим компонентом. Пусть, например, добавляется кнопка `TButton`, расположенная на панели `TPanel`. Наш компонент `TControlManager` тоже должен расположиться на панели `TPanel`, причем в той же позиции, но поверх кнопки. Если в список добавить две кнопки, расположенные на разных панелях, наш компонент `TControlManager` не сможет оказаться одновременно на двух разных панелях, чтобы накрыть кнопки.

Получается, что если в список добавляется компонент, предок которого отличается от предков компонентов, уже находящихся в списке, список очищается и новый компонент остается в списке единственным.

Далее проверяем, пуст ли список. Если пуст, то процесс накрытия достаточно прост. Нужно встать на тот же компонент (или форму), на котором находится добавляемый компонент:

```
Parent := clControl.Parent;
```

Делаем свой компонент видимым и доступным:

```
Visible := True;
```

По умолчанию компонент `TControlManager` невидим. Если в его списке ничего нет, то компоненту незачем находиться на экране и перекрывать что-то своей рабочей областью.

Затем назначаем своему компоненту `TControlManager` размеры добавляемого компонента, но при этом позиция должна быть на два пиксела меньше и на четыре шире и выше, чтобы перекрывать добавляемый компонент с небольшим запасом (запас нужен, чтобы рисовать маркеры выделения):

```
SetBounds(clControl.Left - 2,
   clControl.Top - 2,
   clControl.Width + 4,
   clControl.Height + 4);
```

Если в списке уже есть компоненты, все немного сложнее. Для определения позиции компонента `TControlManager` нужно найти левую верхнюю и правую нижнюю точки описанного вокруг существующих компонентов прямоугольника; например, при выделении двух компонентов наш компонент `TControlManager` перекрывает их так, как показано на рис. 4.10. Пунктирной линией показаны размеры и расположение компонента `TControlManager`.

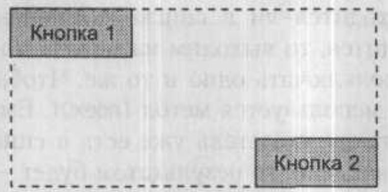


Рис. 4.10. Перекрытие нескольких компонентов

Это немного неудобно, потому что если в перекрытую область попадет другой (не выделенный) компонент, работать с ним будет трудно, ведь доступ к нему закроет наш компонент TControlManager, находящийся поверх всех.

После установки размеров добавляем новый компонент в список с помощью метода Add. Таким образом, сохранив в списке ссылку на компонент, мы сможем с ним впоследствии работать:

```
FControlsList.Add(clControl);
```

Ну и последнее — выносим компонент на передний план, чтобы он оказался поверх остальных, и переносим на него фокус:

```
BringToFront;
SetFocus;
```

Все, компонент добавлен в список. Теперь рассмотрим процедуру очистки — метод ClearControls (листинг 4.16).

Листинг 4.16. Очистка списка компонентов

```
procedure TControlManager.ClearControls;
begin
 FControlsList.Clear;

 Visible := False;
 Parent := nil;
 FDragRect := Rect(0, 0, 0, 0);
end;
```

В первой строке очищаем список, потом делаем компонент невидимым и обнуляем свойство Parent. Таким образом, наш компонент ControlManager перестает перекрывать какие-либо компоненты, «отправляясь в свободное плавание».

Давайте добавим метод Paint, чтобы научить наш компонент прорисовываться. Код этого метода можно увидеть в листинге 4.17. Хотя этот код не полон, все что нужно для понимания алгоритма в нем присутствует.

Листинг 4.17. Рисование маркеров

```
procedure TControlManager.Paint;
var
 aRect: TRect;
 i: integer;
begin
 inherited;

 if FControlsList.Count = 1 then
```

```
    Canvas.Brush.Color := clBlack
   else
    Canvas.Brush.Color := clBtnFace;

   for i := 0 to FControlsList.Count - 1 do
    begin
     // Рисуем маркеры по углам
    end;

  if FControlsList.Count=1 then
    begin
     // Если имеется только один компонент,
     // то рисуем маркеры по центру и краям
    end;
end;
```

Если количество компонентов в списке равно единице, то маркеры будем рисовать черным цветом. Если выделяемых компонентов окажется больше, то рамка у маркеров будет черной, а внутри закрасим их цветом фона кнопки.

Задав цвет, запускаем цикл для перебора всех компонентов. Здесь мы рисуем им угловые маркеры, с помощью которых можно изменять размер компонента. После окончания цикла проверяем, если в списке только один компонент, то рисуем в серединах сторон еще по маркеру; с их помощью можно будет изменять ширину и высоту. Если же в списке несколько компонентов, то изменять размеры нельзя, можно только двигать группу выделенных компонентов, поэтому боковые маркеры рисовать не надо, достаточно угловых, причем закрашенных цветом фона.

Теперь наш менеджер компонентов может добавлять и очищать список компонентов, а также рисовать маркеры выделения. Но если мы попробуем его использовать, он будет просто прятать другие компоненты, поскольку наш компонент пока не прозрачен. Чтобы добавить прозрачность, перекроем метод `CreateParams`. Этот метод компонента `TCustomControl` вызывается, когда нужно задать параметры компонента. В нем мы добавляем новый параметр:

```
Params.ExStyle := Params.ExStyle + WS_EX_TRANSPARENT;
```

Дальше все намного проще. Нужно перехватить три события для нашего компонента `TControlManager`:

- `OnMouseDown` — по нажатии кнопки мыши мы должны начать перемещение компонента;
- `OnMouseMove` — при перемещении мыши мы должны перемещать рамку, которая будет показывать новые размеры и положение компонента;
- `OnMouseUp` — по отпускании кнопки мыши нужно завершить перетаскивание компонента `TControlManager` и расположить все компоненты из списка в новых позициях.

Вроде бы все просто, однако с обработкой события `OnMouseDown` тоже возникает проблема. Дело в том, что это событие генерируется при нажатии кнопки мыши *в любой точке, очерченной на рис. 4.10 пунктирной линией*, в том числе и при нажатии кнопки мыши на компоненте, который не выделен, а просто перекрыт нашим компонентом. Пример подобной ситуации показан на рис. 4.11. Как

видите, из трех имеющихся на форме кнопок мы выделили только две, а третья оказалась перекрытой нашим компонентом.

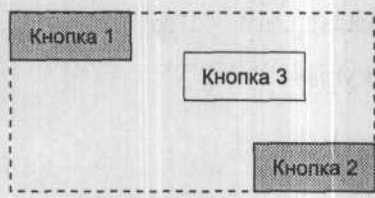


Рис. 4.11. Перекрытие кнопки

Любая попытка щелкнуть мышью на кнопке 3 будет бессмысленной, потому что сообщение от мыши не попадет обработчику событий кнопки 3, а оборвется на нашем компоненте TControlManager. Таким образом, наша задача не просто отфильтровать нажатие кнопки мыши вне кнопок 1 и 2, но и передать это событие обработчику событий кнопки 3 при нажатии кнопки мыши именно на ней.

Итак, давайте посмотрим, как пересылать сообщения, предназначенные перекрытым компонентам. Обработчик события OnMouseDown представлен в листинге 4.18.

Листинг 4.18. Обработчик события нажатия кнопки мыши

```
procedure TControlManager.MouseDown(Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
var
 i: integer;
 aRect: TRect;
begin
 for i := 0 to FControlsList.Count - 1 do
  begin
   aRect := Rect(TControl(FControlsList[i]).Left - 2,
                 TControl(FControlsList[i]).Top - 2,
                 TControl(FControlsList[i]).Left +
                 TControl(FControlsList[i]).Width + 2,
                 TControl(FControlsList[i]).Top +
                 TControl(FControlsList[i]).Height + 2);

    if PtInRect(aRect, Point(Left + X, Top + Y)) then
     begin
      if (Button = mbLeft) then
       begin
        ptDragOffset := Point(X, Y);
        bDragging := True;
        SetCapture(Handle);
        end;
      inherited MouseDown(Button, Shift, X, Y);
      exit;
     end;
  end;

 Cursor := crDefault;
```

```
    SetCursor(Screen.Cursors[Cursor]);
    ReSendMessage(fmDown, Button, Shift, Left + X, Top + Y);
  end;
```

Самое интересное происходит в цикле перебора всех выделенных компонентов. Вначале цикла заносим в переменную aRect (она имеет тип TRect, который удобен для хранения координат прямоугольной рамки выделения) размеры текущего компонента. Затем вызывается функция PtInRect. Ей необходимо передать два параметра:

- размеры и положение рамки выделения в виде переменной типа TRect;
- положение точки нажатия кнопки мыши в виде переменной TPoint.

Функция проверяет, попадает ли точка нажатия кнопки мыши в прямоугольник, и если попадает, то результат становится равным true, иначе false. Получается, что если кнопка мыши нажата внутри рамки выделения, выполняется следующий код:

```
  if (Button = mbLeft) then
   begin
    ptDragOffset := Point(X, Y);
    bDragging := True;
    SetCapture(Handle);
   end;
  inherited MouseDown(Button, Shift, X, Y);
```

Здесь мы проверяем, если нажата левая кнопка мыши, то текущая точка запоминается в переменной ptDragOffset. Раньше для хранения использовались две переменные типа Integer, но в этом примере я решил оставить одну типа TPoint. Далее устанавливаем переменную bDragging в true, что означает начало перетаскивания компонента, и переводим мышь на свой компонент с помощью функции SetCapture. В последней строке вызываем метод MouseDown предка.

Если компонент не найден, то выполняется такой код:

```
  Cursor := crDefault;
  SetCursor(Screen.Cursors[Cursor]);
  ReSendMessage(fmDown, Button, Shift, Left + X, Top + Y);
```

В первых двух строках устанавливаем стандартный указатель мыши — стрелку. Тут нет ничего сложного и интересного. Самое интересное — вызов метода ReSendMessage. Метод достаточно универсален, поэтому его код занимает слишком много места и не может быть приведен здесь (при желании можете найти его на компакт-диске), но о самым важном в этом коде я обязан рассказать.

Назначение метода ReSendMessage в том, чтобы найти компонент, который перекрыт, но не выделен, и передать ему сообщение о нажатой на нем кнопке мыши. Алгоритм работы метода прост.

1. Сформировать переменную типа TMessage. Этот тип используется для передачи сообщений (в данном случае сообщений о нажатии или отпускании кнопки мыши) с помощью функции SendMessage.
2. Запустить цикл перебора всех компонентов, установленных на родителе нашего компонента. Если компонент ControlManager установлен на форме, то будут перебираться все компоненты формы. Тут нужно заметить, что на форме очень часто бывает громадное количество компонентов, поэтому, чтобы цикл

выполнялся быстрее, рекомендую дизайнер располагать на какой-нибудь панели. В моем примере передвигаемые компоненты располагаются на компоненте TScrollBox.

3. Если нужный компонент найден (то есть в точке нажатия кнопки мыши находится невыделенный компонент), то сообщение о нажатии кнопки мыши передается ему, иначе — родителю. В моем примере родитель — это компонент TScrollBox.

Функции MouseMove и MouseUp, которые осталось рассмотреть, также достаточно сложны, и весь код этих функций я приводить не буду. Рассмотрим только логику их работы. Начнем с движения мыши (метод MouseMove).

1. После небольших приготовлений проверяем, сколько компонентов выделено. Если выделен только один компонент, то можно изменять его размеры (когда выделено больше компонентов, изменять можно только положение), поэтому уточняем, где была нажата кнопка мыши. Если в районе левой угловой точки, изменяем тип перетаскивания на dsSizeTopLeft, то есть будем изменять размер, перетаскивая левый верхний маркер; если в правой верхней точке, изменяем тип перетаскивания на dsSizeTopRight, то есть будем изменять размер, перетаскивая правый верхний маркер, и т. д.
2. Определив, в какую сторону пойдет перетаскивание, проверяем, если переменная bDragging равна true, то есть левая кнопка мыши нажата, то изменяем позицию компонента, но не прорисовываем его. Вместо этого вызываем метод DrawDraggingRect, который рисует прямоугольный контур будущего положения. Сам же компонент остается нетронутым.

В методе MouseUp выполняется следующее.

1. Проверяем значение переменной bDragging. Если это значение равно true, значит, имело место перетаскивание, и необходимо перебрать все выделенные компоненты и установить им новые значения положения и размера. Для этого используется метод UpdateChildren.
2. Так как во время перетаскивания мы рисовали прямоугольный контур, его нужно уничтожить. Для этого снова вызываем метод DrawDraggingRect, но в последнем параметре указывается значение false.
3. Освобождаем все захваченные ресурсы:

```
bDragging := False;
SetCapture(0);
Cursor := crDefault;
ReleaseCapture;
```

Если перетаскивания не было, то просто пересылаем событие от мыши компонентам, расположенным под нашим менеджером компонентов. Для этого снова используем метод ReSendMessage.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь компонента находится на компакт-диске в каталоге Sources\ch04\WorkWithComponent.

Использование универсального менеджера компонентов

Давайте посмотрим, как можно использовать описанный компонент. Для иллюстрации я взял пример из раздела «RTTI». Удалите весь код, касающийся перетаскивания. Переменные bDragging, iStartDraggingX, iStartDraggingY в разделе private главной формы больше не нужны. Вместо них объявим только переменную ControlManager типа TControlManager.

Переменная ControlManager является объектом, поэтому перед использованием должна быть создана. Для этого пишем код, выполняющийся по событию OnCreate главной формы:

```
ControlManager:=TControlManager.Create(Self);
```

А по событию OnDestroy главной формы уничтожаем компонент:

```
if ControlManager<>nil then ControlManager.Free;
```

Теперь обработчик события OnMouseDown, который вызывается для компонентов-таблиц (метод NetComponentBoxMouseDown), будет выглядеть так, как показано в листинге 4.19.

Листинг 4.19. Обработчик события OnMouseDown для таблиц

```
procedure TVRDatabaseModelerForm.NetComponentBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 UpdateProperties(Sender);

 // Активация кнопок для работы с выделенным компонентом
 EnableControlsButtons(true);

 // Если не удерживается клавиша Shift, то очищаем список
 if not (ssShift in Shift) then
  ControlManager.ClearControls;

 // Добавляем компонент в список компонентов менеджера
 ControlManager.AddControl(TControl(Sender));

 TTableControl(Sender).SetSelected(true);
end;
```

Для того чтобы наделить компонент, на котором была нажата кнопка мыши, возможностями перемещения и изменения размера, выполняются всего две операции:

1. Проверяем, нажата ли клавиша Shift. Если не нажата, очищаем список компонентов. Таким образом, удерживая клавишу Shift и щелкая мышью на компонентах, можно выделить сразу несколько таблиц.
2. Добавляем компонент, на котором была нажата кнопка мыши, в список менеджера.

Этого достаточно, чтобы можно было перемещать таблицу и изменять ее размер. Запустите программу и убедитесь, что она работает.

ПРИМЕЧАНИЕ

Исходный код для рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch04\WorkWithComponent.

Резюме

Я потратил целый день на создание и отладку универсального менеджера компонентов, зато теперь во всех своих проектах смогу использовать его без изменения кода. Благодаря тому, что я стараюсь писать универсальный код, разработка моих программ идет быстрее.

Однако мой менеджер компонентов не идеален. Его можно и нужно улучшить. Например, если при нажатой клавише Shift дважды щелкнуть на компоненте, то в первый раз он должен выделиться, а во второй раз выделение должно быть снято. У меня при втором щелчке ничего не происходит — после первого щелчка компонент попадает в список выделенных компонентов, а второй щелчок уже ничего не меняет.

Добавьте возможность снятия выделения с компонента, на котором происходит повторный щелчок. Это очень просто — достаточно удалить его из списка. Какой компонент удалять? Вспомните, как мы определяли присутствие компонента в списке (методом IndexOf), и все решится одной строкой кода.

Вариантов для расширения много. Только наращивайте возможности именно компонента, а не своей программы, это пригодится вам в будущем.

Отображение файлов в памяти

Читать и писать файлы стандартными методами не всегда удобно. Во многих случаях приходится сначала загружать файл в память и только потом использовать. Это отнимает очень много времени, а как бы было хорошо иметь возможность сразу обращаться с файлом так же, как с памятью. Ведь ОС умеет это делать. Классический пример — файл подкачки, с которым ОС работает так же, как с оперативной памятью.

В моей книге «Библия Delphi» (я уже не раз ссылался на эту работу) описаны функции, которые я использовал для отображения звуковых файлов в памяти, но там про эти функции рассказано достаточно поверхностно. Сейчас мы рассмотрим их более подробно. Для нашего примера понадобится всего три компонента:

- TOpenDialog — окно открытия файла;
- TButton — кнопка, при щелчке на которой мы будем открывать файл и читать из него данные, как из памяти;
- TEdit — поле, в котором будем отображать прочитанные данные.

Пример кода отображения файла в памяти и чтения его первых двух байтов показан в листинге 4.20.

Листинг 4.20. Пример отображения и чтения файла

```
procedure TForm1.Button1Click(Sender: TObject);
var
 hMapFile: THandle;
 hFile: THandle;
```

```
  lpData: PFileArray;
  wFileSize:Integer;
begin
  if not OpenDialog1.Execute then
   exit;

  hFile:=CreateFile(PChar(OpenDialog1.FileName),GENERIC_READ,
     FILE_SHARE_READ,nil,OPEN_EXISTING, FILE_ATTRIBUTE_ARCHIVE,0);
     wFileSize:=GetFileSize(hFile,nil);

  hMapFile:=CreateFileMapping(hFile, nil, PAGE_READONLY, 0, wFileSize, nil);
  CloseHandle(hFile);
  if (hMapFile=0) then
   begin
    ShowMessage('Ошибка создания файла!');
    exit;
   end;

  lpData:=MapViewOfFile(hMapFile, FILE_MAP_READ, 0, 0, 0);

  Edit1.Text:=PChar(lpData)[0]+PChar(lpData)[1];

  UnMapViewOfFile(hMapFile);
  CloseHandle(hMapFile);
end;
```

Сначала мы должны открыть файл какой-нибудь классической функцией, чтобы получить идентификатор файла. Например, для этого можно использовать функцию `CreateFile`:

```
hFile:=CreateFile(PChar(OpenDialog1.FileName),
   GENERIC_READ, FILE_SHARE_READ, nil, OPEN_EXISTING,
   FILE_ATTRIBUTE_ARCHIVE, 0);
```

Вспомним, какие параметры должны принимать эта функция.

- *Имя файла.* В нашем случае имя будем получать с помощью стандартного окна открытия файлов: OpenDialog1.
- Параметры доступа к файлу: GENERIC_READ (чтение) и/или GENERIC_WRITE (запись). В принципе, мы не будем вызывать функции чтения и записи файла, потому что нас интересует только идентификатор, но я разрешил право чтения. Почему? Если файл открыть только для чтения, то и отобразить в памяти его можно будет только для чтение. Если посмотреть на файл помощи, то там это не является утверждением, но MS рекомендует следовать этому правилу.
- Параметры совместного доступа. В представленном примере используется флаг FILE_SHARE_READ. Он означает, что если другое приложение запросит файл для чтения, то операция пройдет успешно, а для записи файл открыт не будет.
- Указатель на структуру SECURITY_ATTRIBUTES. Если эта структура заполнена, то она определяет, как дочерние процессы могут наследовать идентификатор. Нам наследовать не нужно, поэтому этот параметр равен нулю.

- Параметры создания файла:
  - CREATE_NEW — создать файл, а если он уже существует, то файл не открывать, а вернуть ошибку;
  - CREATE_ALWAYS — создать файл, а если он уже существует, удалить и создать новый пустой файл;
  - OPEN_EXISTING — открыть существующий файл, а если файл с указанным именем не существует, вернуть ошибку;
  - OPEN_ALWAYS — открыть файл, а если файл с указанным не существует, создать файл;
  - TRUNCATE_EXISTING — открыть файл и обнулить его содержимое, а если файл с указанным не существует, вернуть ошибку.
- Атрибуты файла. Их достаточно много, и тут я советую обратиться к файлу помощи по WinAPI. В нашем примере мы использовали атрибут FILE_ATTRIBUTE_ARCHIVE — это атрибут архивного файла.
- Шаблон, на основе которого будет создан новый файл. Нам не нужен шаблон, поэтому этот параметр нулевой.

Так как с файлом мы будем работать, как с памятью, необходимо знать его размер, чтобы не превысить. Когда мы просто читаем файл и достигаем его конца, то любые функции чтения начинают возвращать нулевой объем данных и пустое значение. В случае с памятью очень легко обратиться за пределы заданной области и получить в результате переполнение памяти, сбой стека или что-нибудь подобное.

Чтобы не нарушить безопасность программы и ОС, давайте определим размер файла с помощью функции GetFileSize. Ей нужно передать идентификатор файла и переменную, определяющую верхний байт размера в виде значения типа DWORD, а результатом является нижний байт размера DWORD:

```
wFileSize:=GetFileSize(hFile, wFileSize2);
```

Почему так сложно? Дело в том, что результатом функции может быть только переменная типа DWORD (32 бита), а в эту переменную можно записать число, не превышающее 2 Гбайт. Это конечно же мало, поэтому было добавлено еще 2 байта. Таким образом, полный размер файла равен:

Результат := wFileSize2 << 32 + wFileSize

Результат имеет тип Int64 (целое число размером 64 бита). В своем примере я подразумеваю, что мы будем работать с файлами не более 2 Гбайт — думаю, этого достаточно, поэтому я упрощаю задачу и использую только 32 бита, которые возвращает функция.

Теперь можно отображать файл в памяти. Для этого используется функция CreateFileMapping, имеющая 6 параметров.

- Указатель на файл, который нужно отобразить в памяти. Здесь должен быть идентификатор открытого файла. Вместо идентификатора можно указать число 0xFFFFFFFF (MAXDWORD). В этом случае будет открыт не реальный файл, а выделено место в файле подкачки. Это удобно, чтобы разместить в памяти множество временных файлов.

- Указатель на структуру SECURITY_ATTRIBUTES, о которой мы говорили при рассмотрении функции CreateFile.
- Параметры безопасности. Здесь можно указать один из флагов:
  - PAGE_READONLY — память доступна только для чтения, поэтому при попытке записать в нее данные произойдет ошибка доступа (желательно, чтобы идентификатор файла, который передается в первом параметре, был открыт с флагом GENERIC_READ);
  - PAGE_READWRITE — разрешено чтение и изменение данных (желательно, чтобы идентификатор файла, который передается в первом параметре, был открыт с флагами GENERIC_READ и GENERIC_WRITE);
  - PAGE_WRITECOPY — во время доступа на запись (изменение) создается копия страниц.
- Старшие два байта, определяющие максимальный размер файла (dwMaximumSizeHigh).
- Младшие два байта, определяющие размер файла (dwMaximumSizeLow). Если параметры dwMaximumSizeHigh и dwMaximumSizeLow равны нулю, то максимальный размер эквивалентен текущему размеру файла.
- Строка, определяющая имя объекта отображения (lpName).

После вызова функции CreateFileMapping создается объект отображения. Если результат вызова равен нулю, это означает, что в ходе создания объекта произошла ошибка. В противном случае функция возвращает идентификатор объекта. Мы его сохраняем для получения адреса объекта в памяти.

Теперь, чтобы просмотреть данные, необходимо узнать указатель в памяти, где находится отображение файла. Для этого используется функция MapViewOfFile. Ей передаются следующие параметры.

- Идентификатор объекта. Его мы получили после выполнения функции CreateFileMapping.
- Параметры доступа:
  - FILE_MAP_WRITE — открыть для чтения и записи;
  - FILE_MAP_READ — открыть только для чтения;
  - FILE_MAP_ALL_ACCESS — полный доступ (идентичен параметру FILE_MAP_WRITE);
  - FILE_MAP_COPY — доступ на копирование при записи.
- Старшие два байта начала смещения, с которого нужно просматривать отображаемый объект (dwFileOffsetHigh).
- Младшие два байта начала смещения (dwFileOffsetLow).
- Количество просматриваемых байтов (dwNumberOfBytesToMap). Если указать ноль, будут доступны все данные.

Функция может вернуть ошибку, если запросить невыровненные данные. Например, в ОС, которая работает в 32-битн1ом режиме, все данные выровнены по 4-байтному слову, поэтому если запросить данные с байта 5 по байт 437,

произойдет ошибка. Можно указывать только числа, кратные системному размеру страницы.

Как определить размер страницы? Для этого используется функция GetSystemInfo. Например:

```
var
 si:TSystemInfo;
begin
 GetSystemInfo(si);
 si.dwPageSize - размер страницы
end;
```

Тут очень познавательными окажутся примеры для работы со звуком, которые я описывал в книге «Библия Delphi». Давайте запросим все данные, не обращая внимания на страницы:

```
lpData:=MapViewOfFile(hMapFile, FILE_MAP_READ, 0, 0, 0);
```

Результат выполнения функции сохраняется в переменной lpData. Это не что иное как указатель в памяти. Такой указатель можно использовать в любых функциях для работы с памятью.

В данном случае я типизировал указатель как массив байтов, объявив для этого следующий тип в разделе type:

```
TFileArray=array [0..0] of Byte;
PFileArray=^TFileArray;
```

А переменная lpData имеет тип PFileArray. Можно привести этот тип к строке PChar и читать данные файлы посимвольно. Например, чтобы получить нулевой символ, нужно написать:

```
PChar(lpData)[0]
```

Как видите, нет никаких функций чтения данных из файла. Просто обращаемся к данным как к указателю в памяти. Это очень удобно, но не всегда безопасно, потому что слишком просто выйти за пределы заданной области памяти — в этом случае в вашей программе появится диалоговое окно AccessViolation, которое никого из пользователей не обрадует. Данные можно как читать, так и писать, для этого нужно отрыть идентификатор с нужными правами.

После того как вы закончите работу с данными, в соответствии с хорошим тоном программирования нужно будет закрыть файл и освободить память. Открытый с помощью функции CreateFile файл можно закрывать сразу после вызова функции CreateFileMapping.

Указатель на данные, полученный с помощью функции MapViewOfFile, нужно освободить с помощью функции UnMapViewOfFile. Идентификатор, полученный во время создания отображения (CreateFileMapping), освобождается функцией CloseHandle.

ПРИМЕЧАНИЕ

Исходный код для рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch04\MapFileToMemory.

Печать

В некоторых книгах тему печати относят к наиболее сложным, а некоторые авторы вообще стараются ее обходить стороной. Почему? Намного проще выводить информацию на монитор, чем на принтер или другое печатающее устройство. Когда мы работаем с экраном, разрешение экрана (количество точек на дюйм) фиксировано. В случае с печатающими устройствами разница в их разрешении может быть очень большой. В матричном принтере разрешение равно 72 точки на дюйм, в струйном и лазерном — обычно от 300 до 600, хотя некоторые модели способны поддерживать разрешение более 1000 точек на дюйм.

При разрешении в 600 точек на дюйм ширина страницы моего принтера составляет 6814 точек. Это очень много. Если сравнить это значение с показателями для матричного принтера, то разница будет больше чем в 6 раз, что необходимо учитывать при формировании страницы.

Поэтому квадрат размером в 100×100 пикселов при разрешении в 72 точки на дюйм будет выглядеть достаточно большим, а при разрешении 600 точек на дюйм — в несколько раз меньше. Авторы и программисты не очень любят связываться с разрешением, поэтому уходят от этой темы или привязываются к конкретному принтеру, что вызывает массу проблем при эксплуатации программы.

В книге «Библия Delphi» я показал, как можно вывести изображение, которое будет выглядеть примерно одинаково на разных принтерах. Не буду повторяться, но напомню суть представленного там примера. В том примере я привязывался к разрешению устройства печати. Для этого сначала создавался новый документ с помощью метода BeginDoc объекта принтера, затем документ обновлялся, а потом мы получали его разрешение:

```
Printer.BeginDoc;
Printer.Canvas.Refresh;
PointsX:=GetDeviceCaps(Printer.Canvas.Handle,LOGPIXELSX)/72;
PointsY:=GetDeviceCaps(Printer.Canvas.Handle,LOGPIXELSY)/72;
```

Разрешение можно получить только после создания документа. Дело в том, что принтер может работать с разными разрешениями. Например, максимальное разрешение моего принтера 600 точек на дюйм, но по умолчанию установлено 300. Конкретное значение пользователь может выбрать в окне настройки печати, и выбранное значение закрепляется за документом уже после его создания.

Для определения параметров устройства используется функция GetDeviceCaps. В первом параметре нужно передать указатель на устройство, тогда второй параметр укажет на свойство, которое мы хотим узнать. В данном случае это значения LOGPIXELSX и LOGPIXELSY, которые соответствуют количеству пикселов в ширину и высоту, разделенному на 72. Почему результат делится на 72? Это как бы поправочный коэффициент. Число 72 соответствует значению разрешения матричного принтера, но вы можете выбирать другое значение.

После определения параметров устройства можно выводить на печать изображение. Для этого я всегда использую метод CopyRect, который позволяет масштабировать изображение. В нашем случае масштабирование необходимо, потому

что каждая координата изображения корректируется в соответствии с коэффициентом. Полный текст примера и более подробное описание можно найти в книге «Библия Delphi».

Итак, мы вспомнили, как масштабировать изображения, а теперь разберемся с выводом текста. Сами буквы масштабировать не надо, потому что все TrueType-шрифты векторные, что позволяет легко их масштабировать без потери качества. Проблема в правильной установке пробелов между строками, размеров страницы и т. д. Как действовать в этом случае? Из личного опыта могу сказать, что наилучшим вариантом является привязка к ширине и высоте символа.

Давайте создадим простое приложение, которое будет выводить на печать текст и покажет, как можно универсальными методами выравнивать строки. Для иллюстрации примера я создал форму (рис. 4.12), в которой задаются:

- шапка — располагается вверху листа и выравнивается по его правому краю;
- заголовок — располагается под шапкой и выравнивается по центру листа;
- текст — заполняет центральную часть листа;
- подвал — печатается внизу листа.

Рис. 4.12. Окно программы для универсальной печати

В листинге 4.21 представлен код, который будет выполняться при щелчке на кнопке Печать.

Листинг 4.21. Печать документа

```
procedure TPrintForm.bnPrintClick(Sender: TObject);
var
  iWidth, iHeight, StrMiddle: Integer;
  iTop, i: Integer;
begin
  if not PrintDialog1.Execute then
   exit;

  printer.Orientation:=poPortrait;
  printer.BeginDoc;
```

```
printer.Canvas.Refresh;

  iWidth:=printer.Canvas.TextWidth('A');
  iHeight:=printer.Canvas.TextHeight('A');
  StrMiddle:=iHeight div 3;

  // Вывод шапки
  iTop:=iHeight*2;
  printer.Canvas.Font.Size:=10;
  printer.Canvas.TextOut(printer.PageWidth-
     printer.Canvas.TextWidth(edDocHat.Text)-iWidth*2,
     iTop, edDocHat.Text);
  iTop:=iTop+printer.Canvas.TextHeight('A')+StrMiddle*2;

  // Вывод заголовка
  printer.Canvas.Font.Size:=14;
  printer.Canvas.TextOut((printer.PageWidth-
     printer.Canvas.TextWidth(edDocHat.Text)) div 2,
     iTop, edCaption.Text);
  iTop:=iTop+printer.Canvas.TextHeight('A')+StrMiddle*2;

  // Вывод текста
  printer.Canvas.Font.Size:=10;
  for i:=0 to mmText.Lines.Count-1 do
   begin
    printer.Canvas.TextOut(iWidth*3, iTop, mmText.Lines[i]);
    iTop:=iTop+printer.Canvas.TextHeight('A')+StrMiddle;

    if iTop>printer.PageHeight-iHeight*6 then
     printer.NewPage;
   end;

  // Вывод подвала
  printer.Canvas.Font.Size:=10;
  printer.Canvas.TextOut(iWidth*3, printer.PageHeight-iHeight*4,
     edBottomLine.Text);

  printer.EndDoc;
 end;
```

Обратите внимание, что в этом коде нет конкретных смещений, все вычисляется на основе ширины и высоты букв для текущего разрешения. Так, текст начинает печататься, начиная с позиции `iHeight` (высота буквы), умноженной на два, то есть сверху пропускаем две строки.

Для определения середины текста я использую формулу:

(*ширина страницы – ширина текста*) / 2.

Для деления я использую операцию `div`, которая возвращает результат без остатка. Нам остатки не нужны, поскольку особая точность здесь не требуется.

Для того чтобы выровнять текст по правому краю, я использую формулу:

ширина страницы – ширина текста – ширина двух символов.

Таким образом, справа от печатаемого текста остается пустое пространство в два символа.

Иногда самостоятельно создавать страницы намного удобнее, чем использовать инструменты формирования отчетности (например, Quick Report). Специализированные пакеты хороши для сложных отчетов, а для простых документов их возможности избыточны, и они только усложняют программу и увеличивают размер исполняемого файла.

Предлагаемый мною код позволяет напечатать страницу текста, причем вне зависимости от типа принтера и его разрешения результат должен быть примерно одинаковым. Конечно же, мой алгоритм нельзя назвать идеальным, но он прост, эффективен и достаточен для печати простых документов. Надеюсь, он пригодится вам при разработке собственных проектов.

ПРИМЕЧАНИЕ

Исходный код для рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch04\PrintText.

Сохранение содержимого компонента TTreeView в файле

В последнее время мне часто задают вопрос, как сохранить содержимое дерева TTreeView. Самый простой способ — использовать метод SaveToFile этого компонента. Метод SaveToFile сохраняет в файле содержимое дерева примерно в следующем виде:

```
Программы для работы с графикой
	Adobe
		Adobe Photoshop
		Adobe Premiere
	Microsoft
		Paint
Текстовые редакторы
	MS Word
```

Каждый уровень с помощью символа табуляции визуально отделяется от предыдущего. В данном примере, первый элемент является корневым и содержит две ветки: Adobe и Microsoft. Внутри этих веток содержатся «листья» дерева (рис. 4.13).

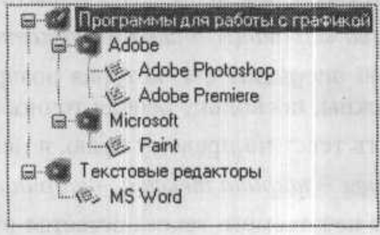


Рис. 4.13. Пример дерева

Таким образом, данные сохраняются одной строкой кода. В этом примере есть «ветки» (папки) и «листья» (программы в папках). Как определить во время загрузки, где папка, а где программа? Я бы сказал, что это невозможно. Первое решение, которое приходит в голову, — крайние элементы представляют собой программы, а любые элементы, у которых есть дочерние элементы, — это папки. Однако это неверное решение, потому что папка не обязательно содержит программы, она может быть пустой. Например, удалите программу Paint из дерева — и папка Microsoft окажется пустой. Сохраните дерево — и после загрузки невозможно будет узнать, папка это или программа, ведь дочерних элементов в папке Microsoft нет.

Мы можем по индексу изображения в списке программно выяснять, папка это или программа. А при сохранении проверять, если в папке нет дочерних элементов, то по умолчанию добавлять в нее какой-то элемент с определенным именем. При загрузке это имя можно не выводить, ведь оно служит только для того, чтобы идентифицировать пустую папку. Такое решение срабатывает, но далеко от идеала и совсем не красиво.

В главе 2 мы познакомились с форматом XML. Этот формат предлагает наиболее эффективное решение проблемы, поскольку, как и дерево TTreeView, поддерживает любой уровень вложенности.

Давайте напишем код, позволяющий:

- эффективно управлять проектами;
- сохранять содержимое дерева в XML-файле;
- закрепить наши навыки работы с форматом XML;
- получить заготовку, которую легко будет превратить в реальную программу.

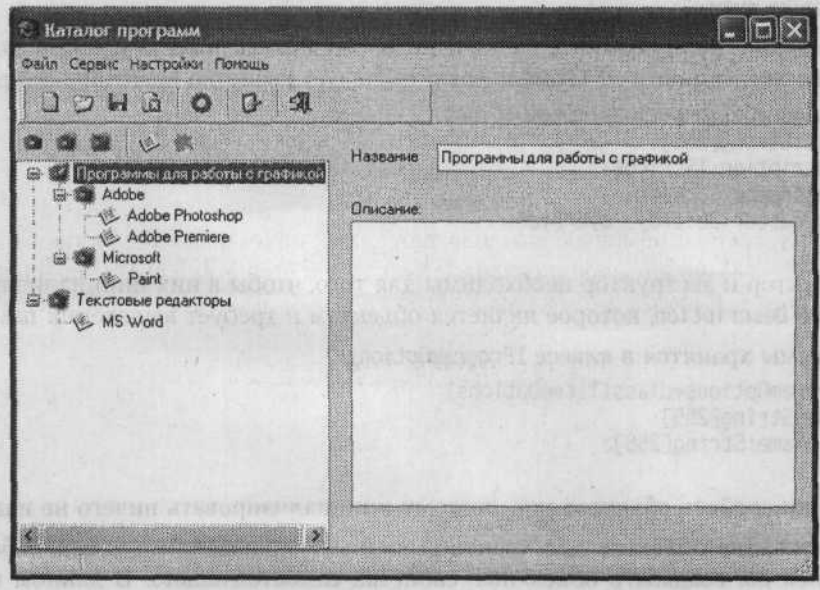


Рис. 4.14. Главное окно будущей программы

В качестве примера я выбрал разработку каталога программ. Очень просто найти применение такому каталогу в реальных условиях, и возможно, этот простой пример вашими усилиями превратится в какой-либо коммерческий продукт. Итак, нам предстоит создать дерево категорий программ, позволяющее добавлять элементы в каждую ветку. Сохранять данные будем в формате XML, который нам уже знаком по главе 2.

Главное окно программы можно увидеть на рис. 4.14. Слева расположено дерево, справа — панель настройки. Точнее сказать, панели две: одна содержит поля для ввода параметров категорий (названий и описаний), другая — для ввода параметров программ (названий и файлов). Панели перекрывают друг друга — в зависимости от типа выбранного в дереве элемента отображается та или другая панель.

Для хранения свойств проекта я завел структуру `TProjectOptionsRecord`:

```
TProjectOptionsRecord=record
 Title:String[255];
 WEB:String[255];
end;
```

Информация о категориях и программах (это элементы дерева) будет храниться в объектах. У обоих типов должно быть какое-то поле, позволяющее однозначно указать, к какому типу относится элемент. Чтобы не делать у двух структур одинаковое поле, я завел базовый класс `TItemOptions`:

```
TItemOptions=class
 ID:Integer;
end;
```

Этот класс состоит всего лишь из одного идентификатора. Если он равен 0 — это папка (категория), если 1 — программа. Таким образом, легко можно создавать новые типы.

Для хранения информации о папке используется класс `TCategoryOptions`, который является наследником от `TItemOptions` и наследует свойство идентификатора:

```
TCategoryOptions=class(TItemOptions)
 Name:String[255];
 Description:TStrings;
 constructor Create;
 destructor Destroy; override;
end;
```

Конструктор и деструктор необходимы для того, чтобы в них инициализировать свойство `Description`, которое является объектом и требует выделения памяти.

Программы хранятся в классе `TProgramOptions`:

```
TProgramOptions=class(TItemOptions)
 Name:String[255];
 FileName:String[255];
end;
```

Тут среди свойств объектов нет, поэтому инициализировать ничего не надо.

Использование объектов для хранения данных упрощает их создание. Без них пришлось бы создавать объектные свойства самостоятельно. В данном случае все будет инициализироваться самими объектами на этапе создания.

Я думаю, что для вас будет несложно добавить в программу возможность создания папок и программ. Это я опущу ради экономии места в книге. Если возникнут проблемы, вы всегда сможете обратиться к исходному коду на компакт-диске. Одно замечание — при создании новой категории или программы создается соответствующий класс (TCategoryOptions или TProgramOptions). Указатель на этот класс нужно сохранить в свойстве Data создаваемого элемента. Так проще будет работать, и впоследствии вам не понадобятся дополнительные контейнеры для хранения указателей на объекты.

Переходим к функции сохранения (листинг 4.22).

Листинг 4.22. Функция сохранения проекта

```
procedure TProgramsCatalogForm.SaveProjectActionExecute(Sender: TObject);
var
 i:Integer;
 Node:TTreeNode;
 r:Boolean;
begin
 if ProgramsTreeView.Selected<>nil then
  ProgramsTreeViewChanging(nil, ProgramsTreeView.Selected, r);

 if ProgectFileName='Noname' then
  begin
   SaveprojectAsActionExecute(nil);
   exit;
  end;

 if FileExists(ProgectFileName) then
  if FileGetAttr(ProgectFileName) and faReadOnly > 0 then
   begin
    Application.MessageBox(PChar(Файл только для чтения '+
        ProgectFileName+'!'), 'Ошибка' ,MB_ICONINFORMATION+MB_OK);
    exit;
   end;

 AssignFile(tfFile, ProgectFileName);
 FileMode := 1;
 Rewrite(tfFile);

 i:=IOResult;
 if i <> 0 then
  begin
   Application.MessageBox(PChar('Не могу сохранить файл '+ProgectFileName+
       ' в папке '+GetCurrentDir+'!'), 'Ошибка' ,MB_ICONINFORMATION+MB_OK);
   Exit;
  end;

 Writeln(tfFile, '<?xml version="1.0" encoding="iso-8859-1"?>');
 Writeln(tfFile, '');
 Writeln(tfFile, '<!— Фленов Михаил (http://www.vr-online.ru) —>');
 Writeln(tfFile, '');
 Writeln(tfFile, '<MENU>');
```

продолжение ⇗

Листинг 4.22 (продолжение)

```
  // Сохранить заголовок
  Writeln(tfFile, '<HEADER>');
  Writeln(tfFile, '<WEB>' + ProjOptions.WEB + '</WEB>');
  Writeln(tfFile, '<TITLE>'+ProjOptions.Title+'</TITLE>');
  Writeln(tfFile, '</HEADER>');

  // Сохранить дерево
  Node:=ProgramsTreeView.Items.GetFirstNode;
  while Node <> nil do
   begin
    WriteCategoryProgramTree(Node);
    Node:=Node.getNextSibling;
   end;

  Writeln(tfFile, '</MENU>');
  CloseFile(tfFile);
 end;
```

Бóльшая часть кода вам должна быть знакома. Просто проверяем наличие имени файла, и если его нет, то вызываем метод, который выводит на экран стандартное окно выбора файла. После этого проверяем, если файл существует, но у него установлен атрибут только для чтения, сообщаем пользователю об ошибке и прерываем сохранение.

Затем открываем файл, очищаем его и устанавливаем режим доступа на запись. Первыми в файл попадают заголовок и информация о проекте. Чтобы проект был более привлекательным и интересным, я наделил его возможностью хранения свойств. Для этого соответствующее окно появляется при создании нового файла (рис. 4.15), а введенные пользователем данные сохраняются в структуре `TProjectOptionsRecord`.

Рис. 4.15. Окно свойств проекта

Я не знал, какими свойствами наделить такой проект, поэтому добавил только поля для ввода названия и URL-адреса. Если вы будете расширять возможности программы, то добавите сюда свои свойства. Свойства проекта сохраняются в XML-файле в теге <HEADER>.

Далее выполняется сохранение дерева. Вот тут кроется самое интересное. Сначала получаем самый первый элемент дерева с помощью метода GetFirstNode и сохраняем его в переменной Node:

```
Node:=ProgramsTreeView.Items.GetFirstNode;
```

Получив этот элемент, запускаем цикл, который выполняется, пока переменная Node не равна нулю, то есть пока не завершится перебор всех элементов первого уровня. Внутри цикла сначала вызываем функцию WriteCategoryProgramTree для записи текущего элемента, а потом в переменной Node сохраняем следующий элемент. Его можно получить с помощью метода getNextSibling. В коде все это выглядит так:

```
Node:=ProgramsTreeView.Items.GetFirstNode;
while Node <> nil do
 begin
  WriteCategoryProgramTree(Node);
  Node:=Node.getNextSibling;
 end;
```

Теперь посмотрим на процедуру WriteCategoryProgramTree (листинг 4.23).

Листинг 4.23. Сохранение текущей ветки

```
procedure TProgramsCatalogForm.WriteCategoryProgramTree(Node: TTreeNode);
var
 ChildNode:TTreeNode;
begin
 if Node=nil then
  exit;

 if TItemOptions(Node.Data).ID=0 then
  begin
   Writeln(tfFile, '');
   Writeln(tfFile, '<CATEGORY>');
   Writeln(tfFile, '<CATEGORYNAME>'+
         TCategoryOptions(Node.Data).Name+'</CATEGORYNAME>');
   Writeln(tfFile, '<CATEGORY_DESCR>'+
           TCategoryOptions(Node.Data).Description.Text);
   Writeln(tfFile, '</CATEGORY_DESCR>');

   // Если у категории есть дочерние элементы,
   // то запускаем цикл их сохранения
   ChildNode:=Node.getFirstChild;
   while ChildNode <> nil do
    begin
     WriteCategoryProgramTree(ChildNode);
     ChildNode:=ChildNode.getNextSibling;
    end;
```

продолжение ☞

Листинг 4.23 (продолжение)

```
      Writeln(tfFile, '</CATEGORY>');
     end;
    if TItemOptions(Node.Data).ID=1 then
     begin
      Writeln(tfFile, '');
      Writeln(tfFile, '<PROGRAM>');
      Writeln(tfFile, '<NAME>'+TProgramOptions(Node.Data).Name+'</NAME>');
      Writeln(tfFile, '<FILENAME>'+TProgramOptions(Node.Data).FileName+
              '</FILENAME>');
      Writeln(tfFile, '</PROGRAM>');
     end;
   end;
```

Код максимально комментирован, чтобы вам проще было в нем разбираться, хотя большая часть и так должна быть вам знакома. Главное — понять алгоритм сохранения.

Все очень просто. Если у текущего элемента свойство `ID` равно нулю — это категория, и в свойстве `Data` текущего элемента находится указатель на класс `TCategoryOptions`. Сохраняем свойства этого класса в XML-файле. Напоминаю, что указатель на класс хранится в свойстве `Data` элемента дерева. Так как это категория, смело приводим этот указатель к типу `TCategoryOptions`.

После этого определяем первый дочерний элемент с помощью метода `getFirstChild`. Если элемент не пустой, то запускам цикл перебора всех дочерних элементов, в котором для каждого из них рекурсивно вызывается функция `WriteCategoryProgramTree`.

Таким образом, данные о дереве сохраняются примерно следующим образом:

```
<CATEGORY>
  <CATEGORY>
    <PROGRAM>
    </PROGRAM>
  </CATEGORY>
  <PROGRAM>
  </PROGRAM>
</CATEGORY>
```

В данном примере показана одна корневая папка, внутри описания которой находится еще одна папка, а также программа. Таким образом, если папка описана внутри другой папки, то и в дереве они отображаются аналогично.

Теперь самое сложное — нужно загрузить XML-файл и восстановить структуру дерева. Строить рекурсивную загрузку сложно, да и не нужно. Обойдемся одной процедурой, которая прекрасно справится с поставленной задачей. Полный код процедуры загрузки можно найти на компакт-диске, а здесь мы рассмотрим только самое основное, что касается восстановления дерева (листинг 4.24).

Листинг 4.24. Загрузка XML-файла

```
function TProgramsCatalogForm.LoadXMLFile(NameOfFile: string): string;
var
 fileend : boolean;
 s       : String;
```

```
  Node    : TTreeNode;
  CurrNode: TTreeNode;
  tagvalue: string;
  ca:TCategoryOptions;
  po:TProgramOptions;
  r:Boolean;
 begin
  // Текущий элемент равен нулю
  CurrNode:=nil;
  ////////////////////////////////////////////////////
  // Здесь мы должны открыть нужный файл для чтения //
  ////////////////////////////////////////////////////

  // Цикл чтения строк
  while not fileend do
   begin
    s:=ReadLine(fileend);

    // Здесь загружаем параметры проекта
    if Uppercase(s)='<HEADER>' then
     repeat
      ////////////////////////////////////////////////////
      // Код загрузки параметров проекта
      ////////////////////////////////////////////////////
     until (fileend) or (uppercase(s)='</HEADER>');

    // Загружаем данные о программе
    if Uppercase(s)='<PROGRAM>' then
     begin
      // Создать объект программы и свойство ID установить в 1
      po := TProgramOptions.Create;
      po.ID:=1;
      repeat
       ////////////////////////////////////////////////////
       // Код загрузки параметров программы
       ////////////////////////////////////////////////////
      until (fileend) or (uppercase(s)='</PROGRAM>');

      // Добавляем элемент в дерево
      node:=ProgramsTreeView.Items.AddChild(CurrNode, po.Name);
      node.ImageIndex:=14;
      node.SelectedIndex:=16;
      node.Data:=po;
     end;

    // Читаем категорию
    if Uppercase(s)='<CATEGORY>' then
     begin
      // Создать объект категории
      ca := TCategoryOptions.Create;
```

продолжение

Листинг 4.24 (продолжение)

```
      ca.ID:=0;
      /////////////////////////////////////////////////////
      // Код загрузки параметров папки
      /////////////////////////////////////////////////////
      until (fileend) or (uppercase(s)='</CATEGORY_DESCR>');
      // Добавляем папку
      CurrNode:=ProgramsTreeView.Items.AddChild(CurrNode, ca.Name);
      CurrNode.ImageIndex:=12;
      CurrNode.SelectedIndex:=17;
      CurrNode.Data:=ca;
     end;
    // Если найден конец папки, то текущий элемент изменяем
    // на один уровень выше
    if Uppercase(s)='</CATEGORY>' then
     begin
      CurrNode := CurrNode.Parent;
     end;
   end;
  /////////////////////////////////////////////////////
  // Закрыть файл, обновить данные
  /////////////////////////////////////////////////////
 end;
```

Я постарался максимально сократить листинг, чтобы показать только самое необходимое. Алгоритм загрузки состоит из следующих основных блоков.

1. Текущий элемент дерева устанавливаем в ноль, потому что проект пустой и в дереве ничего нет.
2. Построчно загружаем данные.
   - Если найдена программа (тег `<PROGRAM>`), то добавляем ее к текущему элементу дерева. Поскольку на этом этапе дерево пустое, программа станет корневой, если так было при сохранении.
   - Если найдена папка (тег `<CATEGORY>`), то читаем данные категории. В цикле чтения мы не дожидаемся появления тега `</CATEGORY>`, который означает конец описания папки. Добавляя новую папку в дерево, изменяем текущий элемент дерева на созданный. Теперь, если будут найдены программы или папки, они добавятся к текущей папке.
   - Если найден закрывающий тег `</CATEGORY>`, это означает, что описание папки закончено и нужно изменить текущий элемент на родительский.

Рассмотрим структуру дерева следующего вида:

```
<CATEGORY> Графика
   <CATEGORY>Adobe
       <PROGRAM>Adobe Photoshop </PROGRAM>
     </CATEGORY>
   <CATEGORY>Microsoft
       <PROGRAM>Paint </PROGRAM>
     </CATEGORY>
</CATEGORY>
```

При загрузке этой структуры наш алгоритм будет работать следующим образом.

1. Текущий элемент равен 0.
2. Найдена категория Графика. В дерево к текущему элементу добавляется новый элемент (текущий элемент пустой, поэтому папка становится корневой), а текущим становится элемент Графика.
3. Найдена категория Adobe; она добавляется к текущему элементу (Графика), а текущим становится элемент Adobe.
4. Найдена программа; она добавляется к текущему элементу (Adobe). Текущий элемент для программ не меняется, поэтому просто идем дальше.
5. Найден тег </CATEGORY>. Он говорит о том, что нужно подняться на один уровень выше, значит текущим элементом становится Графика.
6. Найдена папка Microsoft; она добавляется к текущему элементу и изменяет его.

И так далее.

Вот таким простым способом мы обеспечили возможность универсального сохранения и загрузки дерева. Этот метод весьма эффективный, и я рекомендую по возможности его использовать, поскольку он позволяет сохранять данные абсолютно любого типа и любой структуры.

Даже если вам все стало понятно из описания, я все же рекомендую вам посмотреть код этого примера на компакт-диске. Помимо описанных возможностей я реализовал в коде возможность перетаскивания мышью компонентов внутри дерева.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch04\XMLTree.

Сохранение содержимого компонента TTreeView в базе данных

Сейчас мы рассмотрим пример, в котором данные из TTreeView будут сохраняться в базе данных. Хотя этот пример предназначен для работы с базами данных, я не стал его рассматривать в главе 3, поскольку в основе примера лежит алгоритм сохранения данных компонента TTreeView.

Итак, давайте рассмотрим пример создания дерева TTreeView, данные которого будут сохраняться в базе данных. Для этого примера я не стал использовать MS SQL Server, а взял базу данных Access и в ней создал всего одну таблицу с тремя полями:

- Key1 — ключевое поле, которое имеет тип Счетчик;
- sName — текстовое поле для хранения заголовков элементов;
- idParent — числовое поле для хранения идентификатора главной для данной строки записи.

1. Давайте посмотрим, как хранятся данные в базе данных и как они должны обрабатываться (табл. 4.2).

Таблица 4.2. Пример содержимого таблицы

Key1	sName	idParent
1	Adobe	0
2	Photoshop	1
3	Adobe Photoshop	2
4	Adobe ImageReady	2
5	Microsoft	0
6	MS Office	5
7	Word	6
8	Excel	6
9	Outlook	6
10	Power Point	6

Итак:

- Первая строка с именем Adobe в поле idParent содержит 0. Значит, в дереве эта строка должна быть корневой.
- Вторая строка с именем Photoshop в поле idParent содержит число 1. В таблице с таким же значением в поле Key1 находится строка Adobe, значит, уровень Photoshop должен быть подчиненным по отношению к Adobe.
- Третья строка в поле idParent содержит число 2. В таблице с таким же значением в поле Key1 находится строка Photoshop, значит, уровень Adobe Photoshop должен быть подчиненным по отношению к Photoshop.

Попробуйте мысленно расположить в дереве остальные элементы таблицы. Сравните свои результаты с тем, что показано на рис. 4.16.

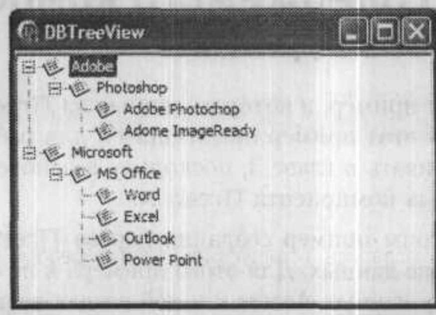


Рис. 4.16. Структура дерева из базы данных

Ниже перечислены элементы управления для главной формы программы, показанной на рис. 4.17.

- Четыре кнопки для выполнения следующих операций:

- □ Обновить — обновить данные из базы данных о состоянии дерева;
- □ Добавить — добавить новый элемент в текущую позицию;
- □ Редактировать — редактировать выделенный элемент;
- □ Удалить — удалить выделенный элемент дерева.

- Компонент TADOTable должен быть соединен с базой данных и нужной таблицей.
- Компонент TImageList предназначен для хранения значков, используемых в меню, панели задач и элементах дерева.

Остальные украшения можно сделать на свой вкус.

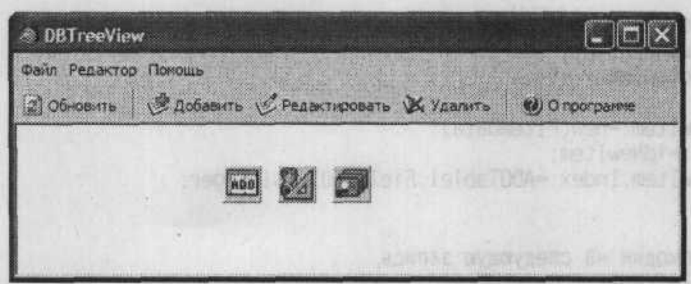


Рис. 4.17. Форма будущей программы

Начнем с загрузки данных. Хотя данных сейчас в дереве нет (можно ввести данные из табл. 4.2, но я этого не делал), их загрузка поможет нам понять, как данные должны храниться. Загрузка данных будет происходить по щелчку на кнопке Обновить и по событию OnShow главной формы. Функция загрузки представлена в листинге 4.25.

Листинг 4.25. Загрузка дерева из таблицы базы данных

```
procedure TDBTreeViewForm.acUpdateTreeExecute(Sender: TObject);
var
  tnNew:TTreeNode;
  idNewItem:PItemData;
  i:Integer;
begin
  // Очищаем текущее содержимое дерева
  TreeView1.Items.Clear;

  // Активируем таблицу
  ADOTable1.Active:=true;

  // Переходим на первую запись и запускаем цикл перебора всех строк
  ADOTable1.First;
  while ADOTable1.Eof<>true do
   begin
    // Переменная tnNew указывает на родительский элемент дерева
    // По умолчанию его делаем нулевым
    tnNew:=nil;
```

продолжение ⇗

Листинг 4.25 (продолжение)

```
    // Если у текущей строки номер родительского элемента больше нуля,
    // то ищем родительский элемент и сохраняем его в tnNew
    if ADOTable1.Fields[2].AsInteger>0 then
     for i:=0 to TreeView1.Items.Count-1 do
      if PItemData(TreeView1.Items[i].Data).Index=
            ADOTable1.Fields[2].AsInteger then
       tnNew:=TreeView1.Items[i];

    // Добавляем текущий элемент в дерево
    with TreeView1.Items.AddChild(tnNew, ADOTable1.Fields[1].AsString) do
     begin
      // Устанавливаем картинки
      ImageIndex:=5;
      SelectedIndex:=5;

      idNewItem:=new(PItemData);
      Data:=idNewItem;
      idNewItem.Index:=ADOTable1.Fields[0].AsInteger;
     end;

   // Переходим на следующую запись
   ADOTable1.Next;
  end;
end;
```

Подробные комментарии в листинге помогут вам разобраться с происходящим. Итак, основа загрузки — цикл, в котором выполняется перебор всех строк таблицы. Если в строке поле с номером 2 (это индекс родительского элемента) больше нуля, то нужно найти в дереве родительский объект и сохранить на него указатель в переменной **tnNew**. Если родительского объекта нет, то эта переменная останется нулевой и при добавлении элемент станет корневым.

При добавлении нового элемента в его свойство **Data** сохраняем указатель на структуру **PItemData**. Эта структура нужна, чтобы в ней сохранить идентификатор строки в таблице базы данных, с которой связан текущий элемент дерева. Сама структура содержит только одно поле — **index**:

```
PItemData=^TItemData;
TItemData=record
   Index:Integer;
end;
```

Если бы в базе данных хранилось больше информации о строке в дереве, то в эту структуру можно было бы поместить и другие поля. А заголовок и так выводится в дереве, поэтому в структуре достаточно хранить только индекс строки в таблице базы данных.

Теперь посмотрим, как можно добавить новую строку в дерево. Код обработчика события находится в листинге 4.26.

Листинг 4.26. Добавление нового элемента в дерево

```
procedure TDBTreeViewForm.acAddTreeItemExecute(Sender: TObject);
var
 Str:String;
```

```
  idNewItem:PItemData;
  tnSelected:TTreeNode;
begin
  // Запоминаем текущий выделенный элемент
  tnSelected:=TreeView1.Selected;

  // Запрашиваем имя нового элемента дерева
  Str:='Новый элемент';
  if not InputQuery('Новый элемент', 'Введите имя элемента', str) then
   exit;

  // Добавляем в дерево элемент и заполняем его свойства
  with TreeView1.Items.AddChild(tnSelected, str) do
   begin
    // Устанавливаем картинки
    ImageIndex:=5;
    SelectedIndex:=5;

    // Создаем структуру для хранения идентификатора
    idNewItem:=new(PItemData);
    Data:=idNewItem;

    // Добавляем в таблицу строку и заносим в нее данные
    ADOTable1.Insert;
    ADOTable1.Edit;
    ADOTable1.Fields[1].AsString:=Str;
    // Если новый элемент не корневой, то сохраняем идентификатор родителя
    if tnSelected<>nil then
     ADOTable1.Fields[2].AsInteger:=PItemData(tnSelected.Data).Index;
    ADOTable1.Post;

    // После сохранения данных записываем в структуру значение
    // ключевого поля новой строки
    idNewItem.Index:=ADOTable1.Fields[0].AsInteger;
   end;
end;
```

Тут все понятно и без дополнительных комментариев. Единственное, что требует пояснения, — зачем в самом начале запоминать выделенный элемент? Дело в том, что дерево не позволяет снять выделение, то есть один элемент всегда остается выделенным. Но как же тогда добавить корневой элемент, ведь для этого нужно, чтобы ни один элемент в дереве не был выделен?

Проблема решается просто. Нужно создать обработчик события `OnMouseDown` для компонента дерева и в нем написать всего одну строку:

```
TreeView1.Selected:=TreeView1.GetNodeAt(X,Y);
```

Здесь мы записываем в свойство `Selected` элемент, который находится в точке щелчка мыши. Для определения элемента, находящегося в точке щелчка, используется метод `GetNodeAt`, которому нужно передать координаты щелчка. Если в этой точке нет элемента дерева, метод возвращает нулевое значение, которое записывается в свойство `Selected`. Таким образом, мы заставили дерево снять выделение программно.

При добавлении элемента запрашиваем его имя с помощью стандартного окна ввода строки функции InputQuery. Тут есть одна проблема. После того как окно закроется, компонент TTreeView получит фокус и снова автоматически выделит один из элементов, а точнее — корневой элемент. Именно поэтому я сохраняю значение выделенного элемента до вывода окна, пока оно еще может быть нулевым.

Реализацию редактирования и удаления данных дерева оставляю вам в качестве домашнего задания. Я только подскажу алгоритм. Тут можно действовать следующим способом.

1. Изменяем заголовок окна. Так как редактируется выделенный элемент, то достаточно записать новое значение в поле TreeView1.Selected.Text.
2. Обновляем содержимое базы данных. Для этого ищем строку с индексом Key1, как у поля index структуры, которая находится в свойстве Data выделенного элемента (PItemData(tnSelected.Data).Index). Искать можно любыми способами, например перебором всех строк, как при загрузке данных.

Самое сложное — реализовать возможность перетаскивания элементов в дереве. Точнее, это не очень сложно, но тут есть один подводный камень. Допустим, вы перетащили элемент Adobe Photoshop на Microsoft и идентификатор текущей строки изменился на 5, как показано в табл. 4.3. После этого при загрузке возникнет проблема из-за того, что таблица просматривается последовательно. Когда мы дойдем до строки 3 с именем Adobe Photoshop, программа попытается найти в дереве элемент с индексом 5, но такого элемента не существует!!! По умолчанию все добавляется в корень (переменная tnNew равна nil), и эта строка станет корневой.

Таблица 4.3. Пример содержимого таблицы

Key1	sName	idParent
1	Adobe	0
2	Photoshop	1
3	Adobe Photoshop	5
4	Adobe ImageReady	2
5	Microsoft	0
6	MS Office	5
7	Word	6
8	Excel	6
9	Outlook	6
10	Power Point	6

Как решить проблему? Самый простой вариант — использовать не таблицу, а запрос (TADOQuery) со следующим SQL-запросом:

```
SELECT *
FROM TreeTable
ORDER BY idParent
```

Запрос вернет нам все строки таблицы, упорядоченные по полю idParent. Таким образом, во время загрузки сначала будут идти все корневые элементы, затем все элементы второго уровня и т. д. Это гарантирует, что при создании очередного элемента все элементы предыдущего уровня окажутся в дереве и необходимый родитель будет найден.

Есть еще один вариант — сохранять данные только при выходе. По событию OnClose очищать содержимое таблицы и, перебирая все элементы дерева (как мы это делали в предыдущем разделе при сохранении дерева в XML-файле), сохранять их в таблице. Но запись в базу данных в любом случае происходит медленнее, чем в файл, поэтому этот вариант я считаю менее предпочтительным.

Попробуйте реализовать все вышесказанное. Всю необходимую информацию я уже дал, осталось только реализовать алгоритм в коде. Я же не стал этого делать, даже на компакт-диске нет соответствующего примера, чтобы у вас не было соблазна подсмотреть.

ПРИМЕЧАНИЕ

Исходный код представленной здесь программы находится на компакт-диске в каталоге Sources\ch04\DBTree.

Глава 5

Управление проектами

В каждой книге я стараюсь дать как можно больше советов и практических рекомендаций. Эта книга тоже не стала исключением, рекомендаций в ней более чем достаточно. Тем не менее осталось еще несколько, которые я хотел бы дать, но которые не подходят по теме ни к одной из предыдущих глав.

Предлагаемые в этой главе рекомендации основаны исключительно на личном опыте. Некоторые из них могут показаться спорными, но я вовсе не настаиваю на том, чтобы вы неукоснительно им следовать. Это всего лишь советы, которые помогают в программировании мне и, надеюсь, помогут вам.

Описываемые здесь рекомендации относятся к управлению проектами, использованию объектов и т. д. В качестве дополнительной информации я рекомендую почитать мою книгу «Delphi в шутку и всерьез: что умеют хакеры», вышедшую в издательстве «Питер» в 2004 году. В ней даны подробные советы по оформлению кода и программ, именованию переменных и объектов и т. д. Не менее полезными могут оказаться советы по оптимизации кода, которые вы также найдете в книге «Delphi в шутку и всерьез: что умеют хакеры». Здесь мы эти темы затрагивать не будем.

Хранение проекта

То, что все относящиеся к проекту модули желательно хранить в одном каталоге, вам должно быть уже понятно. Но в больших проектах этот каталог может стать слишком большим и неудобным в управлении. Как сделать так, чтобы файлов было не слишком много? Давайте для начала посмотрим на форматы файлов, которые создает компилятор Delphi:

- pas – файлы исходного кода модулей;
- dfm – макет визуальной формы модуля;

- dcu — временный файл с промежуточным кодом компиляции модуля;
- ddp — файл свойств модуля.

Все файлы, кроме dcu-файлов, можно хранить в одном каталоге, а вот dcu-файлы лучше убрать в отдельный каталог. Например, пусть ваш проект хранится в каталоге c:\projects\example1. Давайте создадим в этом каталоге папки bin (c:\projects\example1\bin) и dcu (c:\projects\example1\dcu). В первую папку уберем исполняемый файл, а во вторую будем класть dcu-файлы. Обратите внимание, что эти папки необходимо именно создать, иначе при компиляции появятся ошибки, потому что сам компилятор Delphi создавать их не будет.

Перемещать файлы вручную нет смысла. Лучше возложить это на среду разработки. Выберите команду Project ▸ Options и в появившемся окне свойств перейдите в раздел Directories/Conditionals, как показано на рис. 5.1. Здесь есть несколько полей, но из них нам интересны только два.

- Output directory — каталог для выходных файлов. Введите здесь символы .\bin. Теперь исполняемый файл проекта после компиляции будет сохраняться в этом каталоге. Все файлы, с которыми работает программа, также необходимо помещать в папку bin. Это позволит отделить исходный код от других файлов и упростить поставку программы, поскольку чтобы скопировать нужные файлы на целевую машину, достаточно будет просто взять все содержимое каталога c:\projects\example1\bin, а не выискивать нужные файлы среди многочисленных файлов с исходным кодом.
- Unit Output Directory — каталог для выходных модулей. Введите здесь символы .\dcu, и все dcu-файлы будут автоматически сохраняться в этом каталоге.

Рис. 5.1. Раздел Directories/Conditionals окна свойств проекта

Как я уже сказал, если выделить исполняемый файл и все необходимые ему файлы в отдельную папку, то значительно упрощается поставка программ. Достаточно будет скопировать папку bin, и можно быть уверенным, что вы ничего не забыли и не прихватили с собой ничего лишнего.

То, что dcu-файлы хранятся отдельно, дает нам еще одно преимущество. Я надеюсь, вы делаете резервные копии своего жесткого диска и тем более исходных кодов проектов. Если нет, то вам, видимо, еще не приходилось терять данные из-за сбоев ОС или оборудования. Я уже сталкивался с такой проблемой, поэтому резервное копирование в конце месяца стало для меня обязательным.

Чтобы резервная копия занимала меньше места, из нее необходимо исключить dcu-файлы. Действительно, зачем их резервировать, если при перекомпиляции исходного кода эти файлы создаются автоматически. Единственный минус — компиляция без них проходит немного дольше. Но это не страшно, ведь жесткие диски ломаются нечасто, и мы далеко не каждый день восстанавливаем резервные копии файлов с исходным кодом.

Моделирование

Я уже не раз говорил о том, что проекты должны создаваться так, чтобы их проще было сопровождать. Большинство примеров, представленных в предыдущих главах, служили для демонстрации практических приемов написания такого кода, который в дальнейшем может легко расширяться и сопровождаться. В этой главе мы не будем возвращаться к коду, а рассмотрим вспомогательные средства управления кодом и самим проектом.

С недавних пор в Delphi поддерживается моделирование, а в Delphi 2005 прямо в среду разработки интегрирован пакет Borland Together. Единственный недостаток — полноценная поддержка есть только для проектов, ориентированных на платформу .NET. Для классической модели Win32 поддерживается только графическое отображение объектов. Это очень серьезная проблема.

Еще одна очевидная недоработка: пакет Borland Together обладает средствами поддержки таких языков программирования, как Java, Visual C++, C#, VB .NET, а средств поддержки флагманского продукта компании Delphi в нем просто нет!

Самое интересное, что в русскоязычной части Интернета можно найти только минимум информации о UML-моделях. Из-за этого многие программисты просто не знают об их многочисленных достоинствах. Даже скудные возможности моделирования в Delphi могут быть весьма полезными.

Моделирование для Win32

Давайте на практике познакомимся с тем, что предоставляет нам среда разработки Delphi в плане моделирования для платформы Win32.

Для демонстрации возможностей моделей я не буду изобретать ничего нового, а воспользуюсь уже созданным нами проектом из раздела «Сохранение данных проекта» в главе 2. Можете задействовать любой другой проект, особого значения это не имеет. Чтобы начать работать с моделями, необходимо выбрать в меню

команду View ▸ Model View, и перед вами откроется окно просмотра моделей, показанное на рис. 5.2.

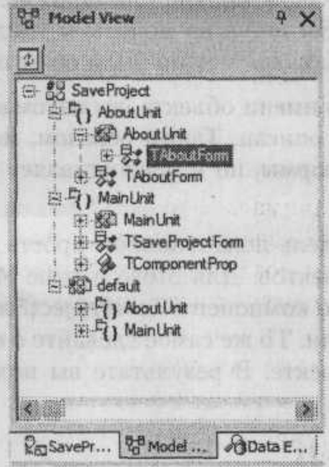


Рис. 5.2. Окно просмотра моделей

В заголовке дерева находится имя вашего проекта (на рисунке это узел SaveProject). Если раскрыть его, вы увидите названия всех модулей, подключенных к проекту. Вы можете создать модель любого модуля своего проекта. Для создания модели щелкните на имени модуля правой кнопкой мыши и в контекстном меню выберите команду Add ▸ Class Diagram. Создайте модель главной формы своего проекта, чтобы на практике можно было пробовать то, что мы будем рассматривать далее.

Самая последняя ветка в дереве имеет имя default. Здесь находится модель, создаваемая по умолчанию, в которой находятся компоненты графического отображения всех форм проекта.

Дважды щелкните на узле default, чтобы увидеть модель, создаваемую по умолчанию (рис. 5.3). На этой модели можно увидеть модули, которые подключены к проекту, и объекты (формы), объявленные в этих модулях.

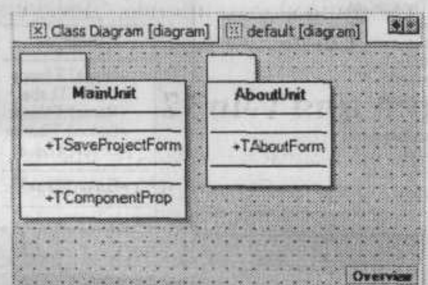


Рис. 5.3. Модель, предлагаемая по умолчанию

Что нам дает такая модель? Во-первых, с ее помощью очень просто найти нужный объект. Например, однажды я отлаживал код модуля, в котором был вызов формы EditPersonForm. По названию видно, что форма создана пользователем, но абсолютно не понятно, в каком модуле она описана. В проекте ничего с похожим названием не было. Взглянув на модель, я увидел, что форма определена в модуле CommonEditUnit. Как еще можно было об этом догадаться?

Если дважды щелкнуть на имени объекта, вы автоматически окажетесь в модуле, в котором этот объект описан. Таким образом, модель позволяет не только рассматривать модули и формы, но и предоставляет удобный механизм поиска нужных данных.

Рассматриваемая нами модель пока слишком проста. Давайте усложним ее, добавив описание самих объектов. Для этого в окне Model View выделите объект главной формы (у меня это компонент TSaveProjectForm модуля MainUnit) и перетащите его на форму модели. То же самое сделайте с остальными объектами, которые присутствуют в проекте. В результате вы получите более сложную модель (рис. 5.4).

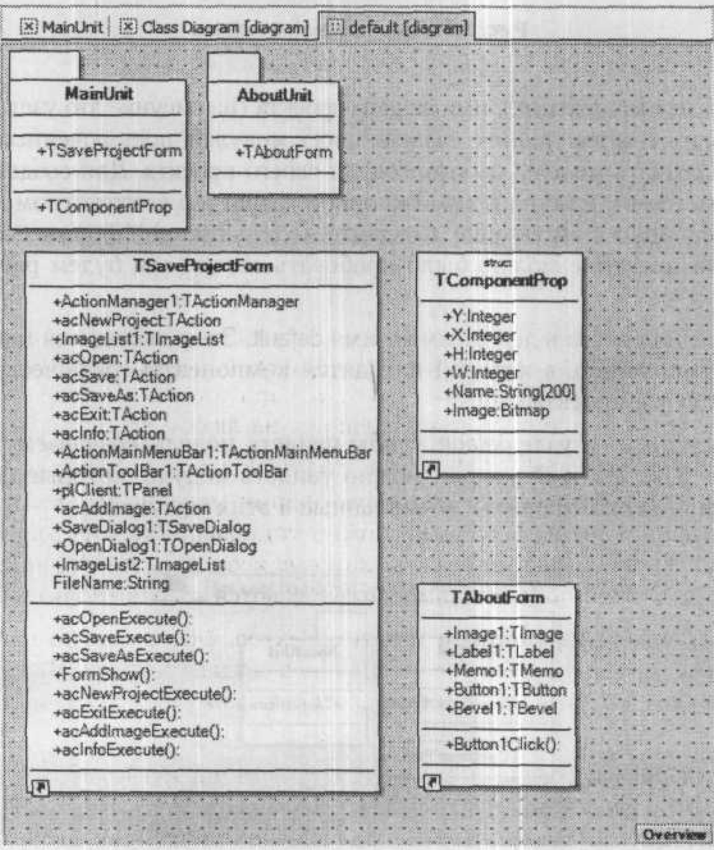


Рис. 5.4. Модель проекта после добавления объектов

Все объекты, создаваемые путем перетаскивания, представляют собой ссылки. Об этом говорит такой же, как у ярлыков Windows, значок ссылки в левом нижнем углу отображений объектов.

Для создания ссылки на объект можно использовать еще один метод.

1. Щелкните правой кнопкой мыши на форме и в контекстном меню выберите команду Add ▸ Shortcuts или нажмите сочетание клавиш Ctrl+Shift+M.
2. Перед вами откроется окно, состоящее из двух списков: расположенного слева иерархического списка объектов проекта (такого же, как в окне Model View) и списка ссылок справа (рис. 5.5). Чтобы создать новую ссылку, переместите соответствующий объект в правый список, щелкнув на кнопке Add.

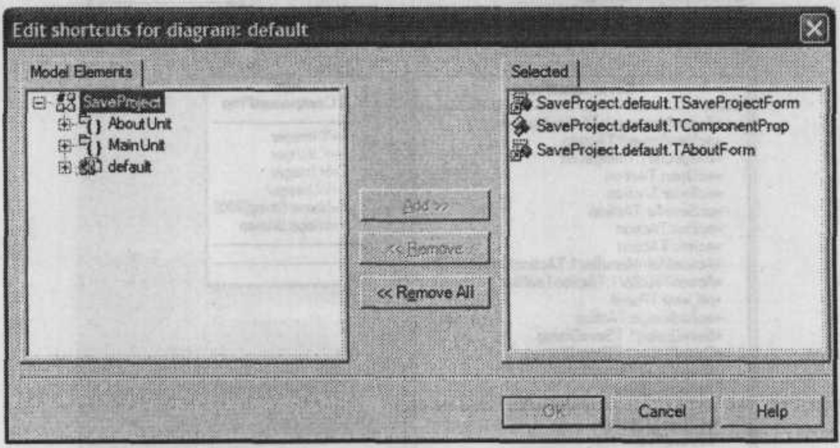


Рис. 5.5. Окно управления ссылками

Отображение объекта для нас более информативно. В заголовке выводится имя объекта, а внутри — его свойства и методы. При этом они разделены по типу доступа — открытые или закрытые. Щелкнув на любом свойстве (или методе), можно перейти в модуль, в котором оно объявлено.

Перетаскивать объекты можно на любую форму модели. Мы создали модель для главной формы, и на нее также можно устанавливать отображения объектов. Я всегда устанавливаю те объекты, которые используются в данном модуле, а в модели, создаваемой по умолчанию, отображаются абсолютно все объекты.

Когда модель содержит слишком много объектов, очень сложно найти то, что нужно. Чтобы упростить работу, я рекомендую создавать связи. В окне UML Class Diagram есть компоненты, с помощью которых можно сделать модель проще и удобнее.

Для создания связей лучше всего подходит компонент Dependency. В модели, предлагаемой по умолчанию, я рекомендую связать модули с отображениями объектов, которые объявлены в этих модулях. Так будет проще найти модуль и его описание.

Для создания связи выделите кнопку компонента Dependency, после этого щелкните сначала на объекте, а затем на модуле, в котором он объявлен. Между двумя объектами модели появится пунктирная стрелка. Создайте связи между всеми объектами модели, и вид формы изменится (рис. 5.6).

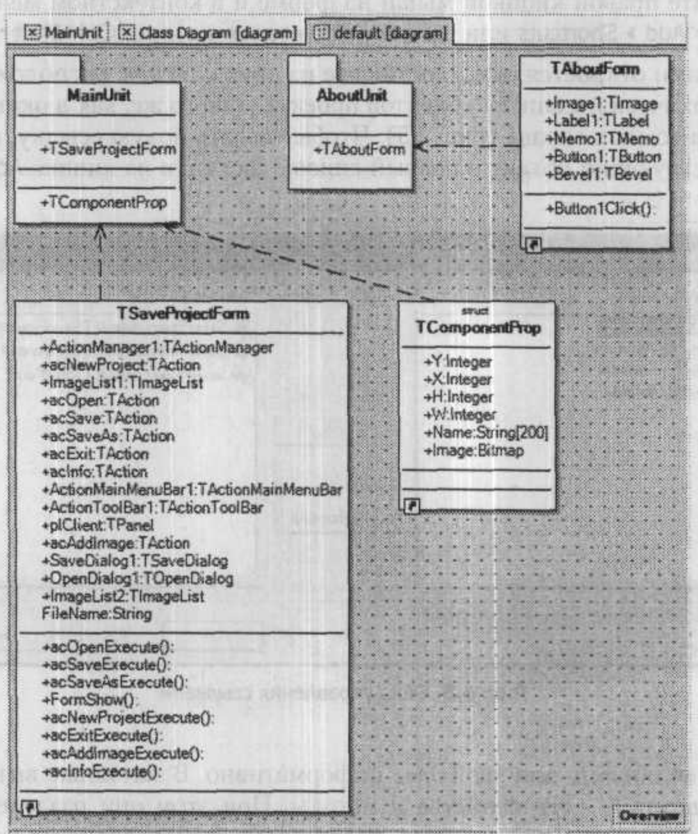


Рис. 5.6. Модель со связями между объектами и формами

С помощью такой модели уже намного проще управлять проектами, особенно если их разрабатывали не вы или над проектом работает целая команда. Если визуально видеть структуру модулей и объектов, да еще и с хорошо проработанными связями, то процесс поддержки проекта значительно упрощается.

В окне модели неплохо было бы оставлять комментарии — для этого в панели UML Class Diagram есть компонент Note. На рис. 5.7 вы можете видеть пример комментария для структуры TComponentProp. Такие комментарии могут упростить жизнь тем, кто будет сопровождать проект.

Чтобы связать между собой комментарий и отображение объекта, необходимо задействовать компонент Note link с панели UML Class Diagram. Он используется так же, как уже описанный компонент Dependency.

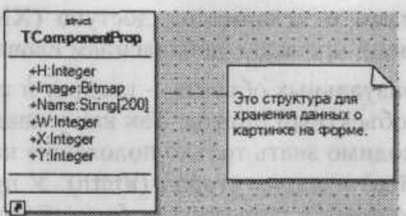


Рис. 5.7. Пример комментария

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch05\Model.

Моделирование для .NET

Средства моделирования для платформы .NET в Delphi более развиты. Чтобы убедиться в этом, создадим новый проект. Для этого выбираем команду File ▸ New ▸ Other и в разделе Delphi for .NET Projects выделяем узел ECO WinForms Application. ECO — это сокращение от Enterprise Core Objects (объекты промышленного ядра).

Прежде чем создать приложение типа ECO WinForms, Delphi запросит ввести имя приложения и указать путь (рис. 5.8). Почему уже на этапе создания нужно указывать путь к папке, в которой должен храниться проект? Просто среда разработки создаст несколько вспомогательных файлов, которые необходимы для правильного моделирования.

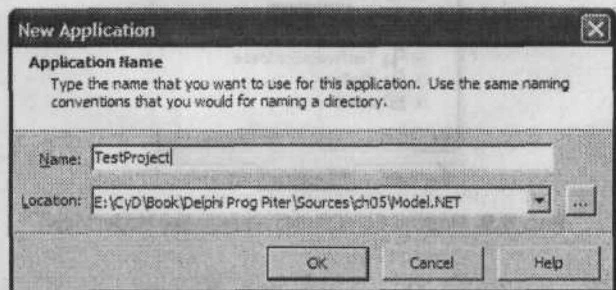


Рис. 5.8. Окно создания нового приложения типа ECO WinForms

Для примера я назвал проект TestProject. Давайте посмотрим, какие файлы создает среда разработки для этого проекта.

- WinForms.pas — стандартное имя файла для WinForms-приложения. Это главная форма, на которой вы можете расставлять визуальные компоненты.
- CoreClassUnit.pas — файл UML-пакета, в котором будут создаваться классы для WinForms-приложения.
- TestProjectEcoSpace.pas — ECO-пространство для приложения. В этом пространстве хранятся объекты приложения во время выполнения. Они обеспечивают

связь между объектами и механизмом доступа (XML или RDBMS). Имя файла состоит из имени проекта с добавлением слова EcoSpace.

Давайте создадим два визуальных объекта — квадрат и прямоугольник. Для этого будем использовать объектный подход. Так как у квадрата стороны равны, то для его описания необходимо знать только положение квадрата (для этого у нас будут свойства `Left` и `Top`) и длину сторон (`Width`). У прямоугольника стороны могут быть разными, поэтому у него должна быть не только информация о положении, но и о длине двух сторон (`Width` и `Height`).

Наделим наши объекты возможностью возвращать площадь геометрической фигуры. Для этого в обоих объектах должен быть метод `GetArea`, который должен возвращать целое число.

Вполне логичным будет создать квадрат «с нуля», а прямоугольник сделать потомком от квадрата, наделив его всего лишь одним новым свойством — `Height`. Давайте так и поступим. Создадим два объекта со всеми необходимыми свойствами, причем один из объектов будет потомком. Для этого нам не потребуется писать ни одной строки кода.

Итак, переходим на вкладку Model View и открываем модель `CoreClasses` (рис. 5.9). Перед вами оказывается пустая форма для создания модели, как и в случае Win32-приложений, но здесь вы можете создавать отображения не только существующих объектов, но и новых.

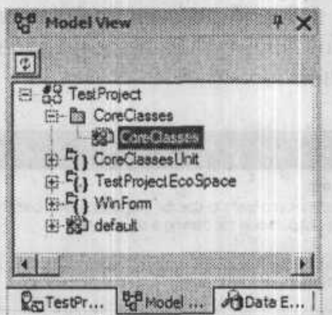


Рис. 5.9. Модель CoreClasses на вкладке Model View

Итак, создаем новый объект. На панели ECO Class Diagram (аналог панели UML Class Diagram, которую мы использовали при моделировании для Win32) выбираем компонент `Eco Class`. Создайте на форме компонент этого типа с именем `TQuad`.

Теперь создадим необходимые свойства. Для этого щелкаем правой кнопки мыши на отображении объекта `TQuad` и в контекстном меню выбираем команду Add ▸ Attribute. В компоненте будет создана новая строка свойства. Назовем его `Left` и оставим для него тип `Integer`, заданный по умолчанию. Если вам потребуется другой тип, необходимо будет полностью написать имя свойства и его тип, например `Left:Integer`. Точно так же создайте свойства `Top` и `Width`.

Теперь создадим метод `GetArea`. Для этого щелкаем правой кнопкой мыши и на отображении объекта `TQuad` и в контекстном меню выбираем команду Add ▸

Operation. В разделе Operation отображения объекта будет создан новый элемент. Напишите в нем GetArea():Integer, чтобы создать нужный метод.

В результате вы должны получить модель необходимого нам объекта (рис. 5.10). Но это только схема, а как же реализация? И это уже сделано. Посмотрите на содержимое файла CoreClassesUnit. Здесь уже есть описание объекта TQuad и созданы все необходимые методы для чтения и изменения свойств. В листинге 5.1. я привел отрывок описания. Не забывайте, что этот код для платформы .NET, и если вы не знакомы с этой архитектурой, не все будет вам понятно.

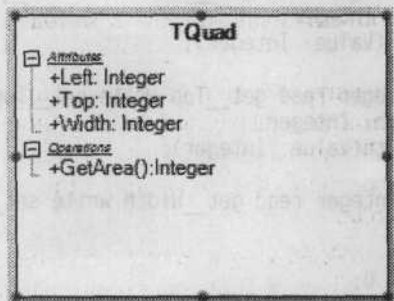


Рис. 5.10. Схема объекта TQuad

Листинг 5.1. Описание объекта TQuad

```
type
  TQuad = class;
  [UmlElement('Package')]
  [EcoCodeGenVersion('1.0')]
  [UmlMetaAttribute('ownedElement', TypeOf(TQuad))]
  CoreClasses = class
  end;

  [assembly: RuntimeRequiredAttribute(TypeOf(CoreClasses))]
  [UmlCollection(TypeOf(TQuad))]
  [EcoAutoGenerated]
  ITQuadList = interface
    function get_Count: Integer;
    function get_Item(index: Integer): TQuad;
    procedure set_Item(index: Integer; Value: TQuad);
    property Count: Integer read get_Count;
    property Item[index: Integer]:
          TQuad read get_Item write set_Item; default;
    procedure Clear;
    procedure RemoveAt(index: Integer);
    function Add(value: TQuad): Integer;
    function Contains(value: TQuad): Boolean;
    function IndexOf(value: TQuad): Integer;
    procedure Insert(index: Integer; value: TQuad);
    procedure Remove(value: TQuad);
  end;
```

продолжение ↷

Листинг 5.1 (продолжение)

```
[UmlElement]
TQuad = class(System.Object, ILoopBack)
strict protected
  [EcoAutoGenerated]
  _content: IContent;
  function get__Left: Integer;
  procedure set__Left(Value: Integer);
  [EcoAutoGenerated]
  property _Left: Integer read get__Left write set__Left;
  function get__Top: Integer;
  procedure set__Top(Value: Integer);
  [EcoAutoGenerated]
  property _Top: Integer read get__Top write set__Top;
  function get__Width: Integer;
  procedure set__Width(Value: Integer);
  [EcoAutoGenerated]
  property _Width: Integer read get__Width write set__Width;
protected
const
  TQuadFirstMember = 0;
const
  TQuadMemberCount = (TQuad.TQuadFirstMember + 3);
strict private
  {$REGION 'Autogenerated ECO code'}
public
  [EcoAutoGenerated]
  function AsIObject: IObject;
  [EcoAutoGenerated]
  constructor Create(content: IContent); overload;
  [EcoAutoGenerated]
  function get_MemberByIndex(index: Integer): System.Object; virtual;
  [EcoAutoGenerated]
  procedure set_MemberByIndex(index:
        Integer; value: System.Object); virtual;
type
  [EcoAutoGenerated]
  TQuadListAdapter = class(ObjectListAdapter, ITQuadList)
  public
    constructor Create(source: IList);
    function Add(value: TQuad): Integer;
    function Contains(value: TQuad): Boolean;
    function IndexOf(value: TQuad): Integer;
    procedure Insert(index: Integer; value: TQuad);
    procedure Remove(value: TQuad);
    function get_Item(index: Integer): TQuad;
    procedure set_Item(index: Integer; value: TQuad);
  end;
  function get_Left: Integer;
  procedure set_Left(Value: Integer);
  [UmlElement(Index=(TQuad.TQuadFirstMember + 0))]
  [EcoAutoMaintained]
  [UmlTaggedValue('derived', 'False')]
  property Left: Integer read get_Left write set_Left;
```

```
    function get_Top: Integer;
    procedure set_Top(Value: Integer);
    [UmlElement(Index=(TQuad.TQuadFirstMember + 1))]
    [EcoAutoMaintained]
    [UmlTaggedValue('derived', 'False')]
    property Top: Integer read get_Top write set_Top;
    function get_Width: Integer;
    procedure set_Width(Value: Integer);
    [UmlElement(Index=(TQuad.TQuadFirstMember + 2))]
    [EcoAutoMaintained]
    [UmlTaggedValue('derived', 'False')]
    property Width: Integer read get_Width write set_Width;
    [UmlElement]
    function GetArea(): Integer;
  strict protected
    [EcoAutoGenerated]
    procedure Initialize(serviceProvider: IEcoServiceProvider);
    [EcoAutoGenerated]
    procedure Deinitialize(serviceProvider: IEcoServiceProvider);
  strict private
    {$ENDREGION}
  public
    [EcoAutoMaintained]
    constructor Create(serviceProvider: IEcoServiceProvider); overload;
  end;
```

Теперь создайте новый объект TRectangle и добавьте к нему свойство Height. Так как объект TRectangle должен быть наследником от TQuad, создадим соответствующую связь. Для этого выполним следующие действия.

1. На панели ECO Class Diagram выберите компонент Generalization/Implementation.
2. Щелкните сначала на отображении объекта TRectangle, затем — на отображении объекта TQuad.

На схеме появится связь в виде стрелки между компонентами TRectangle и TQuad, как показано на рис. 5.11.

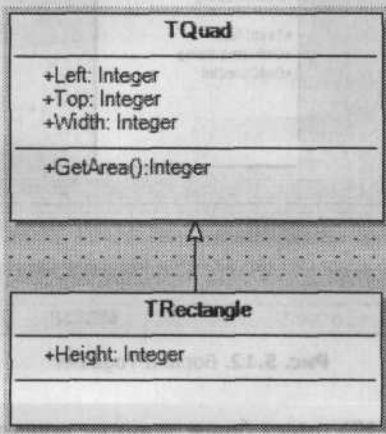


Рис. 5.11. Связь между объектами TQuad и TRectangle

Посмотрите на код файла CoreClassesUnit. Здесь добавилось описание объекта TRectangle. Обратите внимание на то, как он объявлен:

```
TRectangle = class(TQuad)
```

Связь в виде кода также была создана автоматически. Полный код файла я приводить не буду, дабы сэкономить место в книге. Вы всегда можете повторить описанные действия и увидеть все на практике.

Вот так мы создали два объекта, описали все необходимые свойства и методы и при этом не написали ни единой строки кода. Вам остается только наделить метод GetArea логикой, и программа будет готова к работе.

ПРИМЕЧАНИЕ

Исходный код рассмотренного здесь примера находится на компакт-диске в каталоге Sources\ch05\Model.NET.

Резюме

Путем визуального моделирования вы можете сделать код максимально простым и удобным. А для платформы .NET средства моделирования позволяют управлять свойствами и методами объектов без дополнительных усилий. Единственный просчет — в этом случае приходится создавать специализированный проект ECO Application. Это недостаток среды разработки Delphi.

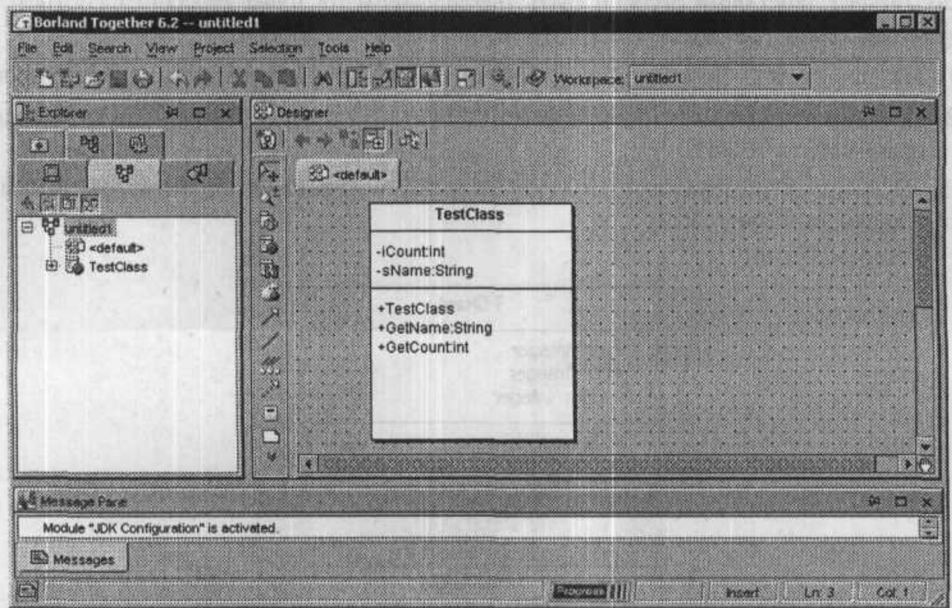


Рис. 5.12. Borland Together

Borland Together (рис. 5.12) имеет более мощные средства моделирования, а результирующий код может сохраняться на языках Java, Visual C++, C#, VB .NET.

Например, код показанной на рисунке модели на языке Java представлен в листинге 5.2.

Листинг 5.2. Класс TestClass, сгенерированный с помощью Borland Together

```
/* Generated by Together */

public class TestClass {
    public TestClass() {
    }

    public String GetName() {
    }

    public int GetCount() {
    }

    private int iCount;
    private String sName;
}
```

Как видите, пакет Borland Together создал новый класс с необходимыми переменными и заглушками для всех функций, которые вы описали визуально.

Будем ждать и надеяться, что в ближайшее время визуальная поддержка будет реализована в Delphi и Win32-проектах. Если бы эта поддержка была уже сейчас, я мог бы экономить кучу времени при работе со структурами данных и создании новых компонентов, ведь визуальное представление намного проще и удобнее.

Заключение

Совершенству нет предела. Большинство книг закладывают базовые знания, а практические навыки приходится приобретать самостоятельно. Я постарался показать вам различные решения типовых задач и направить вашу творческую энергию в нужное русло, а дальнейшее совершенствование может прийти только с опытом.

Не стану утверждать, что мои рекомендации являются единственно верными. Хотя они основаны на большом опыте, это опыт только одного программиста. Чаще всего описанные алгоритмы помогают, но бывают случаи, когда приходится искать другое, более мощное решение. Если вы встретитесь с такой ситуацией, я рекомендую вам следовать приведенным ниже правилам.

- Заранее продумывайте алгоритм работы программы. Его можно написать на листке бумаге, в графическом редакторе или в специализированных программах. Наиболее удобным вариантом я считаю язык UML, но это тема отдельной книги. Что выбрать, сказать трудно. Но ваша цель — увидеть задачу в целом и полностью продумать алгоритм ее решения. Попробуйте учесть все возможные варианты масштабирования (наращивания производительности), чтобы сопровождение не оказалось сложнее разработки. Возможности масштабирования в значительной степени зависят от используемых алгоритмов.
- Старайтесь сразу же создавать универсальный код. Если вы не «программист-однодневка» и собираетесь в дальнейшем писать другие программы, то универсальный код поможет вам в решении ваших задач. Накапливая опыт, собственные компоненты и модули с функциями, вы будете сокращать время разработки программ.
- Если вы пишете универсальную функцию, не храните ее в качестве метода в модуле формы. Лучше сразу же вынести ее в отдельный модуль, который легко можно будет подключить к любому проекту.
- Оптимизируйте код, но только после того, как программа готова. Очень часто одни и те же фрагменты кода приходится переписывать по несколько раз, а каждый раз тратить время на оптимизацию бессмысленно. В книге «Delphi в шутку и всерьез: что умеют хакеры» я посвятил целую главу оптимизации. В ней я говорил, что оптимизировать надо слабое звено программы. Но вы

должны помнить, что самым слабым звеном является алгоритм. Если выбрано неверное направление, то любые попытки сделать код быстрее дадут лишь малую часть того, что может дать выбор более быстрого алгоритма.

- Универсальность и производительность — абсолютно противоположные понятия. Универсальный код не может быть производительным, потому что в нем делается много проверок и он должен работать в любых условиях. Перед написанием программы определитесь, что именно вам нужно, — универсальность или производительность, и в зависимости от этого разрабатывайте алгоритм программы. На первоначальном этапе разработки можно писать максимально универсальные решения, потому что в этот момент сложно предвидеть, что понадобится в окончательной версии. После внедрения программы необходимо переходить к оптимизации, и на этом этапе можно отказаться от универсальности там, где она не нужна, тем самым повысив производительность.

Эти правила позволяют мне создавать проекты, которые легко сопровождать. Например, я четыре года работал в одной крупной фирме, и после увольнения уже в течение двух лет, если просят, я с удовольствием занимаюсь сопровождением своих проектов. Они работают исправно, поэтому вносить изменения приходится достаточно редко, а наращивание функциональности превращается в приятное времяпрепровождение за компьютером в среде разработки Delphi.

Я надеюсь, мои знания и предложенные в этой книге рекомендации помогли вам подняться на ступень выше. Программирование — это область, в которой постоянно приходится развиваться, изучать что-то новое, пробовать и тестировать. В связи с этим хочется закончить эту книгу словами, которыми я начал заключительную главу: *совершенству нет предела*!!!

Флёнов Михаил Евгеньевич

Delphi 2005. Секреты программирования (+CD)

Главный редактор	*Е. Строганова*
Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*В. Шрага*
Литературный редактор	*А. Жданов*
Художник	*Л. Адуевская*
Корректоры	*Н. Соломина, И. Хохлова*
Верстка	*Н. Баланина*

Лицензия ИД № 05784 от 07.09.01.

Подписано в печать 17.08.05. Формат 70×100/16. *Усл. п. л. 21,93.*

Тираж 4000 экз. Заказ № 2579.

ООО «Питер Принт». 194044, Санкт-Петербург, Б. Сампсониевский пр., д. 29а.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.

Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького

Федерального агентства по печати и массовым коммуникациям.

197110, Санкт-Петербург, Чкаловский пр., 15.

В 1997 году по инициативе генерального директора **Издательского дома «Питер»** Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан **«Книжный клуб Профессионал»**. Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области.
Как и все развивающиеся проекты, с течением времени книжный клуб вырос в **«Клуб Профессионал»**. Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в **«Клуб Профессионал»** вам необходимо:

- ознакомиться с правилами вступления в **«Клуб Профессионал»** на страницах журнала или на сайте **www.piter.com**;
- выразить свое желание вступить в **«Клуб Профессионал»** по электронной почте **postbook@piter.com** или по тел. **(812) 103-73-74**;
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект **«Библиотека профессионала»**.

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись на **«Библиотеку профессионала»**. Она для тех, кто экономит не только время, но и деньги. Покупая комплект – книжную полку **«Библиотека профессионала»**, вы получаете:

- **скидку 15%** от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов – дополнительную **скидку 3%**;
- членство в **«Клубе Профессионал»**;
- подарок – журнал **«Клуб Профессионал»**.

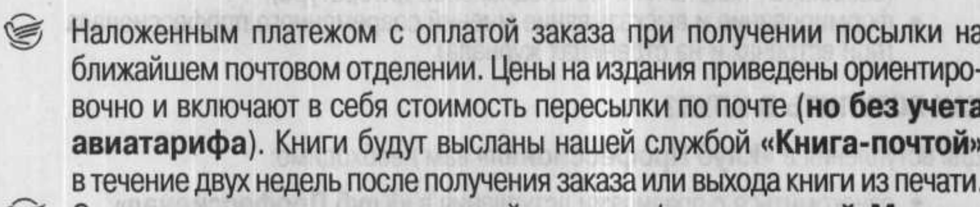
Закажите бесплатный журнал «Клуб Профессионал».

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- по телефону: **(812) 103-73-74;**
- по электронному адресу: **postbook@piter.com;**
- на нашем сервере: **www.piter.com;**
- по почте: **197198, Санкт-Петербург, а/я 619, ЗАО «Питер Пост».**

ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ И ОПЛАТЫ ИЗДАНИЙ:



Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (**но без учета авиатарифа**). Книги будут высланы нашей службой **«Книга-почтой»** в течение двух недель после получения заказа или выхода книги из печати.

Оплата наличными при курьерской доставке (**для жителей Москвы и Санкт-Петербурга**). Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

Вы можете заказать бесплатный журнал «Клуб Профессионал»

ПИТЕР®

Нет времени ходить по магазинам?

наберите:

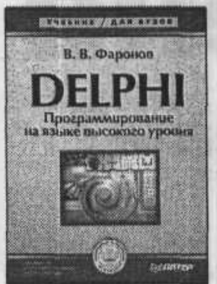
www.piter.com

Здесь вы найдете:

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге — отзывы, рецензии, отрывки
Старые книги — в библиотеке и на CD

И наконец, вы нигде не купите наши книги дешевле!

КНИГА-ПОЧТОЙ



640 с., 17×24, перепл.
Код 1484
Цена наложенным платежом 304 р.

СЕРИЯ «УЧЕБНИК ДЛЯ ВУЗОВ»

В. Фаронов

DELPHI. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ: УЧЕБНИК ДЛЯ ВУЗОВ

Книга посвящена новейшей версии Delphi 7 Studio. Здесь изложены как приемы программирования в среде Delphi, ее главные составные части – галереи компонентов, хранилища объектов, вспомогательный инструментарий, так и сам язык программирования Delphi. Подробно рассматриваются компоненты программ, некоторые дополнительные возможности – динамически подключаемые библиотеки, интерфейсы, технология COM и система ModelMaker. Книга может быть полезна как начинающим – в качестве пособия для первоначального изучения среды и языка Delphi, так и опытным программистам, желающим пополнить свои знания в области применения языка Delphi.
Допущено Министерством образования Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника».

464 с., 17×24, обл.
Код 1174
Цена наложенным платежом 216 р.

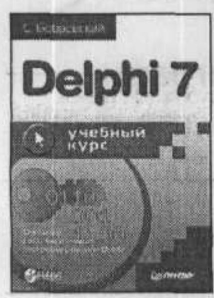
В. Фаронов

ПРОГРАММИРОВАНИЕ БАЗ ДАННЫХ В DELPHI 7. УЧЕБНЫЙ КУРС

В книге описываются многочисленные визуальные и невизуальные компоненты, а также технологии, использующиеся для создания приложений баз данных. Последовательно рассматриваются три наиболее распространенных архитектуры баз данных – файл-серверная, клиент-серверная и трехзвенная. Многие описываемые технологии могут быть применены и в более ранних версиях пакета Delphi. Книга содержит также значительный объем тщательно отобранной и хорошо организованной справочной информации.

С. Бобровский

DELPHI 7. УЧЕБНЫЙ КУРС



736 с., 12,5×20, обл.
Код 783
Цена наложенным платежом 259 р.

В книге рассмотрены возможности системы программирования Delphi 7, описан язык Delphi, рассмотрены визуальные компоненты системы и методы их создания. Особое внимание уделено принципам и практическим приемам создания сетевых приложений для разных архитектур, разработке программ, поддерживающих основные протоколы Интернета, инструментальным средствам организации эффективной работы программистов. Книга не требует специальной подготовки, может быть использована как пособие для изучающих основы программирования и сетевые технологии, а также как справочник по компонентам Delphi и пособие для самообразования.